GEOMETRIC PREDICTION FOR COMPRESSION

A Dissertation
Presented to
The Academic Faculty

by

Lorenzo Ibarria

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August 2007

GEOMETRIC PREDICTION FOR COMPRESSION

Approved by:

Professor Jarek Rossignac, Advisor
College of Computing
Georgia Institute of Technology

Professor Russeau Mersereau
Electrical and Computing Engineering
Georgia Institute of Technology

Doctor Peter Lindstrom
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

Professor Xiangmin Jiao
College of Computing
Georgia Institute of Technology

Professor Greg Turk
College of Computing
Georgia Institute of Technology

Date Approved: 5 June 2007

To Eva, without whose help I could not do much.

PREFACE

Compression is a pervasive technology. It is used when we talk over the phone, when we watch a movie, and when we work with our computers. Compression research is driven by new data and the need for better resolution, accuracy and control of how to represent and retrieve the information. Compression is an active field. This thesis provides a general overview of the technologies used in compression and delves into the area of compression related with Computer Graphics. Several parts of this thesis were developed at the Lawrence Livermore National Laboratory under the auspices of Peter Lindstrom, and part of my research has been funded by DOE Grants.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Source Code Listings

SUMMARY

This thesis proposes several new predictors for the compression of shapes, volumes and animations.

To compress frames in triangle-mesh animations with fixed connectivity, we introduce the ELP (Extended Lorenzo Predictor) and the Replica predictors that extrapolate the position of each vertex in frame i from the position of each vertex in frame $i-1$ and from the position of its neighbors in both frames. For lossy compression we have combined these predictors with a segmentation of the animation into clips and a synchronized simplification of all frames in a clip.

To compress 2D and 3D static or animated scalar fields sampled on a regular grid, we introduce the Lorenzo predictor well suited for scanline traversal and the family of Spectral predictors that accommodate any traversal and predict a sample value from known samples in a small neighborhood.

Finally, to support the compressed streaming of isosurface animations, we have developed an approach that identifies all node-values needed to compute a given isosurface and encodes the unknown values using our Spectral predictor.

CHAPTER I

OUTLINE

This thesis is about the compression of animated graphical models, i.e., low-dimensional sampled geometric data that exhibits spatial (and possibly temporal) coherence. Specifically, we investigate three common types of data:

1. A scalar field sampled on a regular grid in 2D or 3D, and animations of such scalar fields over time.

2. A sequence of triangulated surfaces with constant connectivity that represent the ordered states (key-frames) of an animation sampled over time.

3. A sequence of isosets (isocurves in 2D and isosurfaces in 3D) of a constant scalar field defined by isovalues selected interactively by an operator interested in exploring a particular range of the scalar field. Evolving the isovalue corresponds to animating the isoset.

The main thrust of our contribution, and the unifying theme of this thesis, is geometric prediction, which is an essential component of geometry compression. The thesis is divided into four parts, which we summarize below.

1.1 Role of prediction in geometry compression

Most of the general data compression techniques exploit statistical bias in the symbol frequency to compress the data stream by using shorter codes for more frequent symbols or groups of symbols. Graphical models store values of scalar fields or point (vertex or control point) coordinates in space (and time). Typically these values are taken from a large interval and quantized to the nearest representable value in the chosen format (for example, floating point or integer). When high accuracy is desired, repetitions are unlikely and hence the statistical bias is weak. When accuracy loss is acceptable, stronger value quantization

may be used to reduce the alphabet and hence enhance the statistical bias. Unfortunately, some of the applications we cater for, such as scientific simulation, engineering analysis, or medical visualization, manipulate values represented with high precision (such as 32-bit integers or floating point values) and often require lossless compression. Regardless, most geometry compression approaches use prediction to replace the original values by corrections. The use of prediction improves the statistical bias because, with good predictions, the corrections are concentrated around zero. The prediction techniques investigated in this thesis estimate a value or location from the values or locations of previously transmitted neighbors. Thus, the order of data transmission (traversal) restricts which predictors can be used.

We explore three traversals:

1. **Scanline** transmission transmits the data one slice after another, in the order in which the data is stored on disk, computed, or acquired. It is well suited for **streaming**, which is essential when only a fraction (slice) of the data may be accessed at any time during compression or decompression.

2. **Hierarchical** transmission sends an approximation first (for example through averaging or subsampling) and then refines it by transmitting compressed upgrades. It allows the user to inspect the initial or successive approximations and to decide when and where more accurate versions are necessary. Because the initial approximation corresponds to an undersampling of the data, the accuracy of predictions is low in the initial phases, but increases with subsequent refinements.

3. **User-driven** transmission lets the user select one-by-one which lower-dimensional subset (slice, key-frame, or isoset) is to be transmitted next. As such, it affords combinations of scanline and hierarchical transmission, but also allows the user to progressively refine desired subsets.

Typically, scanline transmission uses the same predictor for all values, with the exception of initialization for border samples. Hierarchical transmission usually requires a larger,

but fixed set of predictors. The user-driven transmission requires a general prediction mechanism because the configuration of known neighbors (stencil) is arbitrary.

Throughout this thesis, we propose several predictors that are affine combinations of neighbors (in which the weights sum up to 1, but can be negative) making our predictors affine invariant (independent of the choice of a coordinate system). A predictor is defined by its stencil, which specifies which neighbors are used and associates them with non-zero coefficients. An important property of a predictor is its power, which we define as complexity of polynomials that are perfectly predicted. We explore 1D, 2D, 3D, and 4D stencils and discuss their predictive power and their benefits for the compression of different types of data.

1.2 Scalar fields on regular grids

The first part of our work focuses on regular samplings of scalar fields. These are represented by the values of the scalar field at the nodes of a regular Cartesian grid. For example, in 2D, the scalar field could represent a digital elevation model (height-field or terrain), or the grey-level or color channel of an image. In 3D, the scalar field may represent the distribution of temperature, pressure, velocity magnitude, or density throughout space. A 4D scalar field usually represents the evolution of a 3D scalar field over time and is represented as an ordered collection of 3D slices.

These datasets tend to be large. For example, the Piecewise Parabolic Method simulation (Figure 1) produced by scientists at the Lawrence Livermore National Laboratory is a 4D dataset with $2048 \times 2048 \times 1900$ 3D slices and 270 time steps, requiring 2 Tbytes.

Representations based on regular grids afford algorithmic simplicity and benefit compression because the sample locations are implicit and hence need not be transmitted. Their drawback lies in the fact that the sampling is not adaptive. Hence, to achieve a sampling frequency that may be necessary in one part of the space of interest, smooth portions of space end up being grossly oversampled. Fortunately, the cost of transmitting the values in these relatively oversampled regions may be drastically reduced by a good prediction. Hence, it is vital to seek the best possible predictors for these smooth and usually large

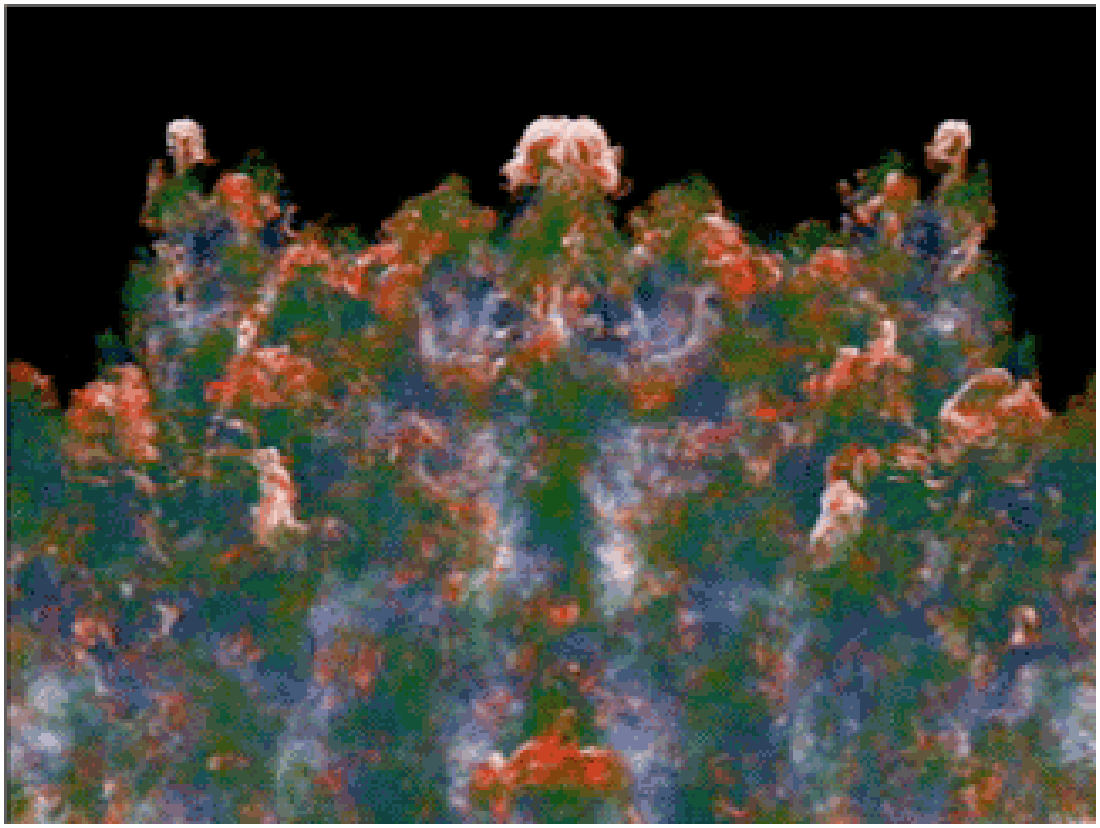


Figure 1: Piecewise Parabolic Method (PPM) 4D data set from a simulation of two fluids interacting. This image is courtesy of Lawrence Livermore National Laboratory.

oversampled portions of space.

For the scanline transmission of n-dimensional scalar fields, we have developed the **Lorenzo** predictor. The 2D parallelogram predictor predicts the value of a node from its three neighbors. The Lorenzo predictor extends the parallelogram predictor to higher dimensions. The Lorenzo predictor has many advantages. It is trivial to implement. It uses a very small stencil (an n-dimensional cube of which one node has an unknown value). Its power ensures that we can predict exactly all polynomials of degree n-1. In fact, one can show that it is the best predictor for such a small neighborhood. We have applied it to various 3D and 4D datasets, achieving a compressed file size that is between 20% and 30% of the size of the uncompressed dataset. Different datasets were compressed using scanline traversal coupled with Lorenzo predictor, and the residuals were encoded with an arithmetic encoder. Compression results vary with the nature of the data, such as the coherence

between neighboring spatial and temporal samples because the sampling rate of the scalar field impacts prediction accuracy, as well as the degree of quantization for the scalar field.

When the scalar field is smooth relative to the sampling density, a higher degree predictor may improve compression. We have developed a second-degree Lorenzo predictor, which we call the **bi-Lorenzian** predictor. It performs the Lorenzo prediction on residues of Lorenzo predictors for the neighbors of the predicted sample. In 2D, the stencil of the bi-Lorenzian predictor is a 3x3 neighborhood and its power is biquadratic, meaning that it can predict exactly biquadratic polynomials that do not have an x^2y^2 term. For smooth data, such as scientific fluid simulations (vorticity, diffusivity, velocity) from the Lawrence Livermore National Laboratory (LLNL), the bi-Lorenzian predictor reduces the error magnitude by 50% with respect to that of the Lorenzo predictor.

Through the application of the Lorenzo predictor on each corner of a 3x3 cube, we produce the **Radial** predictor. The Radial predictor is an interpolating predictor with the same predictive power as the bi-Lorenzian predictor, but due to its symmetry, it outperforms the bi-Lorenzian predictor, reducing the magnitude of the error by an additional 10%. The Radial predictor assumes that all points of the 3x3 neighborhood are known except the center. Although the Lorenzo and bi-Lorenzian predictors each suffice to compress a dataset, the Radial predictor does not. It can only be applied to predict a sample for which all neighbors are known. Hence, the Radial predictor is suited for hierarchical traversal in combination with other predictors, such as predictors that use lower dimensional neighborhoods (one dimensional cubic interpolation) or predictors with larger masks. The set of predictors chosen to complement the Radial predictor define the type of hierarchical traversal, and are generally less effective than the Radial predictor itself.

Driven by the need to complement Radial with adequate predictors for a wider variety of stencils, we have developed the Spectral predictors. We extend Isenburg et al's work on polygons [50], where they compute a Fourier decomposition for polygons of different degrees and assume the higher frequencies to be zero for predicting missing points around the polygon. We construct the spectral predictor to define the m known points in the stencil using m basis functions in order to represent most of the low-frequency response, leading

to small correctors for the missing high frequency content. Thus, Spectral predictors are particularly effective for smooth datasets. We have computed all the spectral predictors for a 3×3 neighborhood in 2D, totaling to 768 different predictors. Deriving the set of Spectral predictors is computationally expensive, but it only needs to be done once. The complete table of spectral predictors has been posted online. The derivation method proves Spectral predictors optimal. In fact, the Lorenzo, bi-Lorenzian and Radial predictors are special cases of the Spectral predictor. Combining Spectral with Radial in lossless hierarchical transmission, we achieve between 10 to 20% reduction in average error magnitude than Radial combined with a low dimensional predictor.

1.3 Animated triangle meshes

The second part of our work focuses on compressing consecutive keyframes of a 3D animation, where each keyframe is represented as a triangle mesh, all the meshes share the same connectivity and the correspondence between the vertices of one mesh and the next is known. Although these assumptions limit the generality of our work, they are common in Computer Animation. For example, predefined motions for characters, obtained from motion capture or through user design, are stored as a sequence of triangle meshes, each one representing the character at a time interval. Early animations had few frames, seldom more than one hundred, and the triangle meshes contained fewer than ten thousand triangles. Current animations have seen an increase in triangle mesh size, with meshes having hundreds of thousands of triangles, yet the length of the animation has increased at a slower pace due to the fact that it is easier to allocate more resources to highly detailed meshes and break animations into small pieces.

A triangle mesh is represented by a table of vertex locations, each represented by the numeric values of its coordinates, and by the connectivity, which is a list of triangles, each defined by 3 vertex indices. There are two major aspects of the compression of triangle meshes: Connectivity Compression and Geometry compression.

Connectivity compression has received a considerable amount of attention from the 3D graphics and modeling community, because connectivity dominates storage cost. Most

Connectivity Compression approaches eliminate the need to compress the vertex indices by reordering the vertices according to a prescribed topological traversal of a triangle spanning tree in the mesh. The only thing that remains to encode is the information of the tip vertex of each triangle: Is it a new vertex or a vertex on the boundary of the previously encoded portion of the mesh? If it already is on the boundary, which one is it? A variety of clever encoding schemes have been developed to compress the answers to these questions. Some exploit the regularity of vertex valences, which are highly biased around 6. We use the Edgebreaker compression, which guarantees that the connectivity of any zero-genus triangle mesh with t triangles can be encoded with 2t bits. Edgebreaker encodes the connectivity as a string of symbols from the set $\{C, L, E, R, S\}$. Connectivity storage cost can be reduced to about 1T bit in practice by exploiting the bias in the frequency of these symbols and even further by using the connectivity and geometry of the neighbors to predict the next symbol, encoding only a verification bit (which is often 1) and occasional corrections.

Given that connectivity can be compressed to only about 1 bit per triangle and that we need to transmit only one connectivity (since it is shared by all meshes in the animation), the main thrust of our work is focused on compressing the geometry.

The geometry (vertex location) of a single mesh is compressed by predicting the tip vertex of each triangle (one-by-one in the order in which these vertices are encountered by the traversal of the triangle-spanning tree) and by encoding the coordinates of the correction vector. Again, if the prediction is good, the correction coordinates are biased towards zero. Typically, each tip vertex is predicted using the parallelogram rule (mentioned when we were discussing the Lorenzo predictor) from the 3 vertices of a previously encoded neighboring triangle. Although more elaborate predictors have been explored, their benefits over the parallelogram predictor are small and uneven.

Instead of encoding the geometry of each mesh (keyframe) independently of the others, we have developed an approach called **Dynapack** that predicts the location of each vertex not only from its neighbors in its keyframe, but also from its position and the position of its neighbors in the previous keyframe. This approach is well suited for **streaming** an animation, since the coder and the decoder only need to access two consecutive keyframes

at a time.

Specifically, we have developed two predictors.

1. The **Extended Lorenzo Predictor** (abbreviated ELP) is an extension of the parallelogram predictor to incorporate coherence over time. Basically, it is a parallelogram predictor on vertex velocities. ELP is a perfect predictor for meshes (or subsets) undergoing an animation that is a pure translation. It requires only seven additions per vertex, which makes it a perfect candidate for hardware implementation.

2. The **Replica** predictor is computed by expressing each vertex in terms of **relative coordinates** with respect to the parent triangle (the one used for the parallelogram prediction). Then, in the next keyframe, the same relative coordinates are used to predict the vertex with respect to the evolved version of the same parent triangle. Replica is more expensive to compute than ELP (requiring several dot products and square roots), but may improve compression, since it is a perfect predictor when the animation of the mesh (or of a subset) can be modeled by an affine transformations (translation, scaling, rotation).

We have evaluated these two predictors against each other, against the parallelogram (space-only) and against a linear predictor of the trajectory or velocity of each vertex (time-only) on several animations. For example, in the "Chicken Run" animation, ELP achieved 3.01 bits per coordinate and Replica achieved 2.91 bits per coordinate when the vertex locations were initially quantized to 13 bits per coordinate. Both predictors were 50% better than the space-only parallelogram predictor or the time-only predictor. These results emphasize that for some animations it is important to exploit both spatial and temporal coherence. However, on the datasets available to us, we did not notice a strong benefit of Replica over ELP.

Although more drastic quantization tends to improve compression, it considerably reduces the accuracy of the animation. For example, the animations we tested should not be quantized to less than 11 or 12 bits per coordinate if they are to be visualized for close viewing. Hence, to increase compression while attempting to preserve accuracy, we have

explored the use of triangle mesh simplification as a form of lossy animation compression.

Simplification collapses one edge at a time, hence merging two vertices and removing two triangles. A variety of heuristics have been proposed to select at each step the edge whose collapse will have the smallest adverse effect on the accuracy of the resulting mesh. A popular technique estimates the cost of each collapse by using the sum of the squared distances from the new location of the two vertices to the planes that contain their original incident triangles [33]. This approach is effective for eliminating vertices from smooth oversampled portions of the mesh.

Unfortunately, simplifying each keyframe independently of the others using this error minimization heuristic will in general produce meshes that no longer have the same connectivity, hence requiring that the connectivity of each mesh be transmitted. Furthermore, as we have pointed out earlier, effective connectivity compression schemes reorder the vertices of the mesh. Hence, using them would destroy the correspondence between the vertices of consecutive keyframes, and would make interpolation (for in-betweening or slow motions) extremely difficult.

To overcome this problem, we have developed a synchronized simplification, which collapses the same edge on all keyframes. The difficulty here was mainly in the efficient estimation of the overall error produced by such a global edge-collapse, measured on all keyframes. The advantage of the approach is that we still have the same connectivity. The drawback is that some portions of the mesh may be smooth and hence oversampled during part of the animation, but may be curved during other parts. The synchronized simplification will associate them with a high overall error, and hence will prevent their simplification, reducing the effectiveness of the approach.

We have explored a compromise, where we split the mesh into short clips (consecutive runs of keyframes) and simplify each clip using the above synchronized simplification. Then we transmit each clip by encoding its constant connectivity and using ELP to predict the vertex locations from the previous keyframes in the clip, hence exploiting the space and time coherence. The challenge here was to decide where to split the animation and how to reduce the encoding of the connectivity of each clip. We have developed the **Clippacker**

solution, which uses a greedy heuristic to split the animation into clips. We propose to use a greedy method to find a balance between the extra cost of encoding the connectivity for each clip and the savings more clips allow for simplification. It builds clips incrementally from single meshes until the addition of a single mesh is not cost effective.

We have also explored several approaches to the encoding of the clips connectivity and the correspondence of vertices. We propose a method to encode the simplification process and the decimations that a mesh undergoes to become a simpler mesh. This information allows two different sets of triangles to relate to each other, thus improving prediction, geomorphs and the mapping of attributes from vertex to vertex. We achieved 1.3 bits per triangle of the original connectivity compression. Although this is not competitive with the encoding of the individual connectivity, it may be important if one needs to transfer texture coordinates or other vertex attributes between clips or if one wishes to use geomorphs for producing animated meshes that interpolate between the last keyframe of a clip and the first keyframe of the next clip.

Testing this approach on real datasets has proven delicate because the animations available to us were designed carefully not to oversample the model unnecessarily. Hence, simplification had little benefit. Nevertheless, our solution is comparable (within 20%) to the best global compression techniques that perform Principal Component Analysis on the entire animation and hence require a significant amount of space and time. We conclude that due to its simplicity, effectiveness, speed and suitability for streaming, the Clippacker-Dynapack combination may offer the best solution for applications where animations must be streamed.

1.4 Exploration of ranges of isosets

The third part of this thesis focuses on ranges of isosets. The isoset $S(t)$ of a scalar field F is the set of points P for which $F(P) = t$. We say that t is the isoset value. The isosets of a terrain are the isoclines. The isosets of a volumetric model (regular sampling of F on an axis-aligned grid of nodes) are its isosurfaces. A particular isosurface may be extracted by visiting all the cells of the volume or more efficiently by starting from a set of seeds and

invading each connected component using a process similar to the Edgebreaker traversal discussed above. The extraction process typically builds a triangle mesh that approximates the isosurface. Hence, one can use the triangle mesh compression techniques described above to transfer such isosurfaces.

Our primary objective is to provide a remote operator with an effective tool for inspecting the results of large simulations. Typically we envision that the operator will not be satisfied with viewing one or several isolated isosurfaces, but will want to explore selected ranges of isosurfaces, animating them by smoothly varying the isoset value over the range.

Transmitting a dense sampling of independently compressed isosurfaces is too expensive. Transmitting only a few keyframes is not acceptable because their different connectivity would make it difficult to compute an interpolating animation, especially because the operator typically wants to be certain that no phenomenon of importance can be lost by such a temporal sampling.

Hence, we have focused our efforts on transmitting not the isosurfaces, but the nodes necessary to produce the animations of isosurfaces over the desired range. In particular, our approach does not require the operator to select a range a priori. Instead, the operator may select an initial isovalue and then extend the range in either direction at will or start a new range by requesting a different isovalue. We transmit only the values that are necessary to start or extend the range and that are not yet known to the decoder. Since we do not know a priori which values are needed, we predict which values are needed and correct the prediction if necessary. The desired values are encoded by using the Spectral Predictor discussed above because the stencils for each new value vary widely, depending on which neighbors have been already transmitted.

For each vertex of the mesh of the isoset, there are on average 1.5 nodes to be encoded in 2D, decreasing to less than one node per vertex in 3D (the nodes are shared by several vertices).

In 2D, using our approach each node has to be encoded with full precision to correctly determine the isocurve. The cost of encoding a single isolated isocurve through its nodes is given by the need to encode the nodes determining the isocurve. Because there are 50%

more nodes than vertices, our approach has an overhead cost of 1.5. The cost of encoding the curve through lossless curve compression methods is proportional to the number of vertices.

However, isosets with close isovalues might share nodes, or the previously encoded nodes can appear in the stencil for prediction of new values. Our technique is able to exploit this locality and significantly reduce transmission cost when the isovalues are close.

We tested our method in a series of datasets, for example the vorticity dataset and a distance field. For the distance field, the compression of a single isoset results in 12 bits per node, 17 bits per vertex of the isoset. When the encoded isosets had overlapping nodes, we obtained 9 bits per node, 12 bits per vertex up to 5 bits per node, 1.8 bits per vertex. The order of transmission of isosets influences the compression; in a test encoding 9 isosets hierarchical encoding resulted in a 30% increase in file size from the sequential encoding due to the low prediction accuracy present in the first levels of the hierarchy. On complex datasets, such as the vorticity data from LLNL, a single isoset obtained 20 bits per node and 33 bits per vertex. When encoding an overlapping set of isosets the compression improved to 16 bits per vertex, for increments of 1.0 in isovalue. We have extended the method to 3D, the compression of a synthetic dataset achieved 9.4 bits per node and 5.22 bits per triangle. A range of isosets that were spaced in increments of 2.0, lowered the bits per node to 7.5, the bits per triangle to 4.1. A spacing of 1.0 reported 4.3 bits per node and 1.5 bits per triangle. The prediction is more accurate in 3D, although we only used a 2D axis-based version of Spectral prediction.

In summary, we have proposed several techniques for the compression and streaming of graphical models. Our techniques are based on prediction; new predictors are the main focus of our contributions. We have proposed predictors for 2D, 3D and 4D stencils. Our predictors are aimed for a variety of different traversals, are simple to implement (linear combination of points), take advantage of the multi-dimensional coherence in data, and are optimal in the context of their neighborhood. The combinations of a traversal and a set of predictors result in efficient compression of data with a compact memory footprint, suitable for streaming progressive decompression or interactive exploration.

1.5 Chapters description

Chapter 2 presents the types of data for which we propose methods: regular grids, animated meshes and isosets. It continues with describing the principles that govern compression such as entropy, which measures the limits of compression of a stream of data. Finally this chapter covers the objectives of our approaches: compression, streaming and resource economy.

Chapter 3 provides a survey of compression algorithms that have inspired our approaches. Amongst the most prominent are Arithmetic Encoding, Run Length Encoding, JPEG, K-Means and wavelets.

Chapter 4 is dedicated to the compression and processing of regular grids representing 2D or 3D scalar fields and their animation. We explain our compression methods and compare different strategies to traverse a regular grid. We introduce the following predictors: Lorenzo L^1, bi-Lorenzian L^2, Radial R and the Spectral Predictors S. We provide algorithms that couple our predictions and traversals to produce methods that compress the grid values.

Chapter 5 reviews compression and simplification literature of triangle meshes including: Edgebreaker, Progressive Meshes and Quadratic Error Simplification. Our approach in animated mesh compression builds on these methods; we include them here for completeness.

Chapter 6 presents our contributions to the compression of 3D animations. We describe our Dynapack and Clippacker approaches for the lossless and lossy compression of animated triangle meshes.

Chapter 7 reviews the literature in isoset extraction and compression. We describe extraction (marching cubes, seeds), compression, transmission, visualization and progressive encoding.

Chapter 8 explains our approach for the interactive exploration of regular grids using isosets. We detail our approach for the cases of 2D and 3D, and describe isoset encoding, regular grid traversal and prediction applied to partially known regular grids.

Chapter 9 summarizes the approaches presented in this thesis, discusses their strengths

and weaknesses, and provides insights for future work directions.

The contributions this thesis presents are the *ELP* and *Replica* Predictors for animated mesh compression, the *Dynapack* technique for the lossless compression of animated triangle meshes, the *Clippacker* technique for the lightweight lossy compression of animated meshes, the *Lorenzo* Predictor for the compression of Volumetric data, and related predictors such as the *bi-Lorenzian* and *Radial* Predictors. Another one of our contributions is the *Spectral* Family of predictors, that are optimal for smooth datasets and can be applied in any situation. We propose an innovative system for the exploration of isosurfaces, both in 2D and 3D.

The work in this thesis is the result of a collaboration with Professor Jarek Rossignac from the Georgia Institute of Technology and Dr. Peter Lindstrom from the Lawrence Livermore National Laboratory.

CHAPTER II

BACKGROUND

2.1 Data

We first discuss the domain of our work and the associated data structures. A significant portion of this thesis deals with scalar fields sampled over regular grids; we simply call them *grids*. A grid is defined as a set of samples from a scalar field in nodes of a regular lattice. The lattices used in the grids discussed in this thesis are rectilinear and have uniform sampling density along each axis.

A *node* is a sample on the grid. There is a value associated with each node. A *cell* is a cube bounded by 8 nodes in 3D or 4 nodes in 2D. Four nodes make a cell in 2D, eight nodes make a cell in 3D. If the grid were to be represented with lines through the nodes, each segment connecting two nodes would be what we call an *edge*.

Regular grids do not need to store any spatial information about the nodes, which is implicit in the way a regular grid is defined. On the other hand, a regular grid typically stores values for all nodes in the grid. Seldom all the values of the grid have valuable information since users only desire to explore a part of the grid. To increase the resolution of nodes in a section of the grid, all the grid has to increase in resolution, often creating oversampled areas in the grid.

Storing a floating point scalar value per node would require a volume cost in bits:

$$\text{Volume Cost} = 32 * N^3 \tag{1}$$

In physical simulations each sample can have several values, such as pressure, velocity and density [23]. For example, a 1024 cubed volume data needs 4 Gbytes of space, which is only one single time frame of a simulation. The PPM dataset (Figure 5) is a scientific simulation with dimensions $2048 \times 2048 \times 1900 \times 270$. It needs close to 2 Terabytes of disk space. These volumetric datasets may be visualized in several ways. A common technique

is to take cross-sections of the volume [83]. Cross-sections lack the 3D context needed to reveal the structure of the data. An alternative is to use volumetric rendering, which integrates opacity and reflections along each view (Figure 2 [65]). Finally, the volume may be explored via isosurfaces, an accurate representation of the set of nodes in the volume sharing a property. As such, isosurfaces contain volumetric information, but several isosurfaces are required to explore the progression through space indicated by the volume.

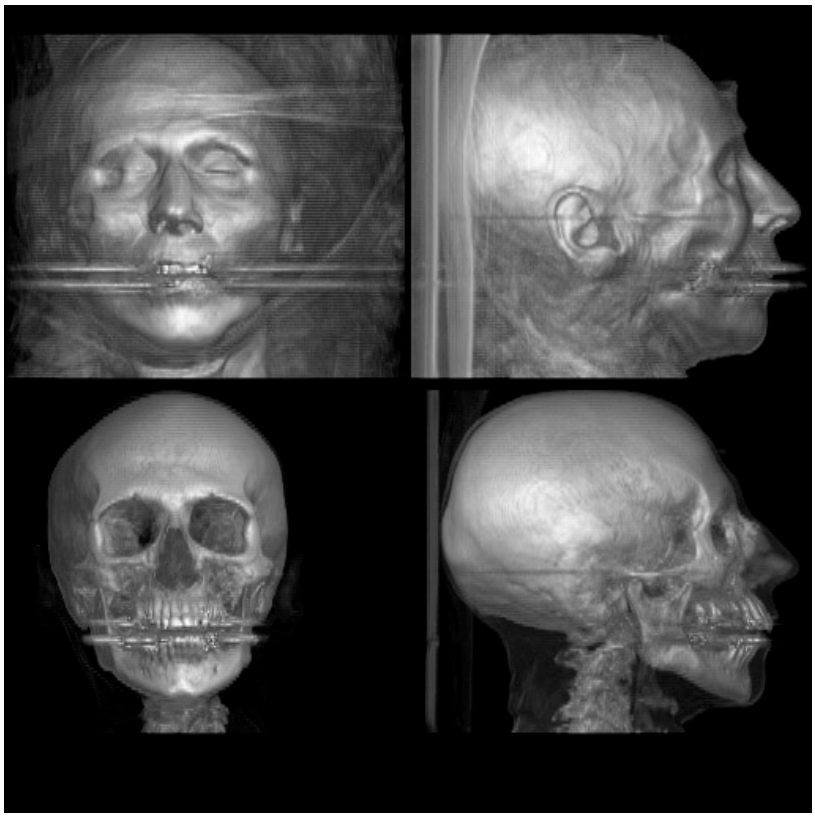


Figure 2: Cadaver head, volume data from the Standford volume data archive [65]. The figure has been generated with volume rendering technique to show the outer part of the skull.

Meshes are a common construct to store graphical data. A mesh is composed of simple shapes, such as triangles, quads, generic polygons and tetrahedra. In this sense, a mesh is a specific case of a simplicial complex. "The geometry of a triangulated model is denoted as a tuple (K, V) where the *abstract simplicial complex* K is a combinatorial structure specifying the adjacency of vertices, edges, triangles, etc., and V is a set of vertex positions specifying the shape of the model in $\mathcal{R}^3$. More precisely, an abstract simplicial complex K consists of

a sequence of vertices $\{1, \ldots, m\}$ together with a set of non-empty subsets of the vertices, called the *simplices* of K, such that any set consisting of exactly one vertex is a simplex in K, and every non-empty subset of a simplex in K is also a simplex in K." Popovic and Hoppe [84]. This thesis focuses on meshes composed of triangles. In contrast to regular grids, triangle meshes allow us to have more dense sampling on areas of higher interest, such as areas with high curvature, and have low sampling on areas of not much interest, like flat areas. However, the positions of each vertex must be encoded. Regular grids and meshes do not represent the same information.

The geometric part of a triangle mesh representation can be represented by an array of vertices $\mathbf{V}$. The incidence, represented as a list of triangles, is stored in the $\mathbf{T}$ array (for example, see Rossignac's Corner Table [89]). In order to store the incidence for each triangle, T stores the three indices for each vertex for each triangle. T and V are the only arrays needed to represent a triangle mesh. To be able to traverse the triangle mesh, it is useful to precompute the adjacency relationship between triangles. We build array $\mathbf{O}$, which contains the precomputed adjacency relationship between triangles. Two triangles share an edge if they have two vertices in common. Therefore, array T is enough to determine triangle adjacency of a mesh; however a search must be performed to determine each adjacency. For each triangle array O stores the indices of the three triangles that share an edge with it. O can be efficiently computed from T with cost $n \log n$. O can accelerate queries of neighboring elements from linear cost to constant cost.

"Topology is the study of properties of figures that endure when the figures are subjected to continuous transformations" [41]. Since triangle meshes are sampled surfaces, developers of compression algorithms must be careful when modifying triangle meshes to preserve topology if this is desired. It is required in many applications that topology remains constant. A *connected component* in a surface is the set of points such that there is a path between any pair of points that does not leave the connected component. An intuitive notion is that each disjoint part of the surface is a connected component.

We work with triangle meshes that are *manifold*. For all points in a manifold mesh, the intersection between the mesh and a sphere of infinitessimal radius centered at the point

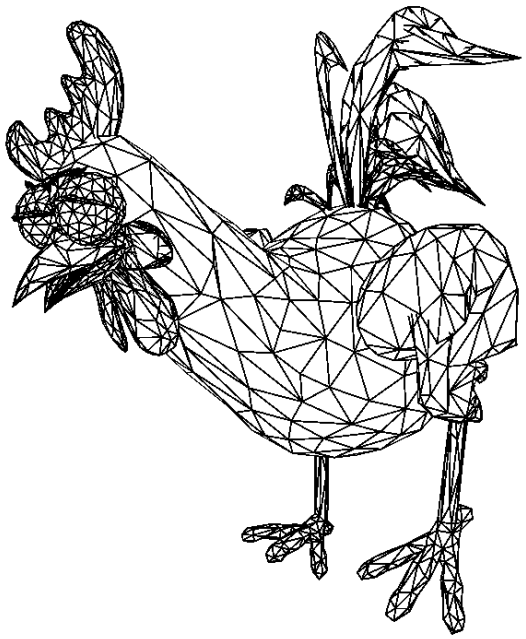


Figure 3: Rendering of a triangle mesh using wireframe mode, the individual triangles are shown.

is homeomorphic to a disk. Non-manifold triangle meshes require an extension to data structures and require special attention to those cases in the algorithms. The genus of a surface is defined as the maximum number of closed simple cuttings without disconnecting the surface.

The triangles in a mesh define a water tight surface. The easiest way to animate a triangle mesh is to have a sequence of triangle meshes, and display one at each frame, like movies display one picture every $\frac{1}{24^{th}}$ of a second. As long as the discrepancy between consecutive meshes is small, the sequence of their rendering produces a smooth animation. Often animations have the same connectivity on frames to reduce the storage size and to reduce popping artifacts due to changes in connectivity.

Grids are used to store the sampling of a scalar function $f(x)$ in space. An isoset is the set of points that fulfill a property, $f(x) = v$ with v being a particular isovalue of interest, which determines the isoset.

$$S_v = \{p | f(p) = v\} \tag{2}$$

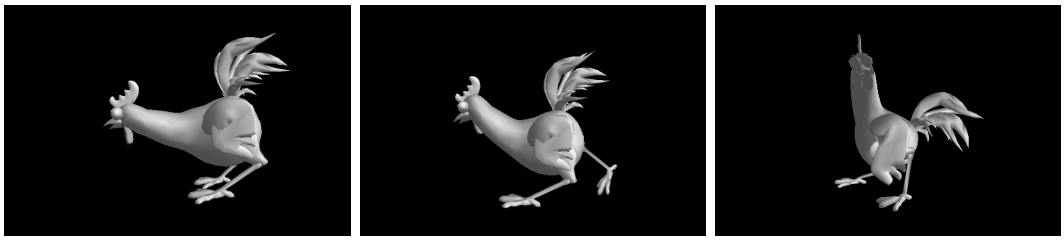


Figure 4: Chicken Crossing animation, from Microsoft and J. Lengyel. All the frames are independent triangle meshes that have the same connectivity, their sequence is defined as an animation.

For a function representing the temperature over a portion of space, an isoset is the surface where all the points have the same temperature, a temperature front. These surfaces are water-tight unless chipped by the boundaries of the volume. Isosets that are extracted from volume data [71, 78, 79] are commonly represented as triangle meshes. Often isosets have a large number of connected components and a large number of handles, and might not be manifold (even though it is possible to make them manifold). The added complexity makes isosets slightly more costly to encode than a regular triangle mesh; yet for all purposes we consider the cost of encoding an isosurface the cost of encoding its mesh representation.

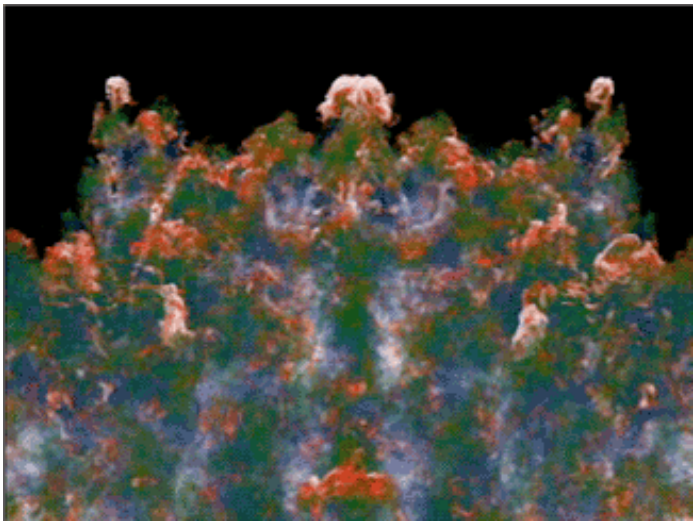


Figure 5: Rendering of the PPM dataset, from the Lawrence Livermore National Laboratory. The PPM dataset is a physical simulation of the interaction between two fluids after a shock. The figure represents the surface dividing the space into one fluid and another in the middle of the mixing process. It shows high complexity.

One single isoset might not be enough information for the user. If the value used to

extract the isoset is not correct, or if the user wishes to browse nearby values, new isosets have to be generated. To address this problem, isosets can be extracted in batches, sampling an interval of isovalues. The set of isosets sampling an interval of isovalues is called an IsoSetRange:

$$\{S_v | v \in [v_1, v_2]\} \tag{3}$$

Because isosets in a range have typically different connectivity, one must either transmit the connectivity of each isoset separately, or develop compression techniques to encode connectivity changes. Both approaches are costly. To alleviate this cost, we encode the points from the regular grid that suffice to produce all isosets in the range.

2.2 General Compression

Compression transforms one representation of information into another while preserving the content but reducing the size. The size is measured in bits. Information **encoding** is the process that takes information in one representation and outputs it in another representation. Compression is one kind of encoding, but there are others that focus on security or redundancy rather than compactness.

In the compression paradigm there are two roles, the *encoder* and the *decoder* role. The encoder has access to the data in its original form and produces a set of bits that the decoder can use to recover the original data. In predictive compression in particular, the encoder must be able to simulate the prediction the decoder will use for each point.

Most information to be compressed is composed of a set of symbols. The number of possible symbols Σ is finite. The length of the information input is not required to be finite; such case would be a compression program streaming data from a server. The trivial encoding of a set of symbols uses $\lceil \log_2 |\Sigma| \rceil$ bits to encode each symbol. Such an encoding is known as raw encoding. Raw encoding is an upper bound in compression and often it is used as the baseline to compare the compression ratio. It has several drawbacks from a compression point of view. If $|\Sigma|$ is 33, each symbol needs 6 bits to be encoded, however only 33 symbols out of 64 are used. Almost half the representation space is wasted. Grouping

symbols into short words may reduce the wasted space. For example, each pair of symbols can be encoded using 11 bits instead of 12 bits. If a symbol is used very frequently while others are seldom used, using 6 bits each time the frequent symbol is encoded is a waste as well.

Variable length coding assigns codes with shorter length to more probable symbols or groups of symbols. This raises the question of what the limit of compression is for a set of symbols with a given probability distribution. Shannon's information theory [96] states that the information in a channel or string of symbols, also known as **entropy** is:

$$H(A) = -\sum_i p(a_i) \log_2 p(a_i) \tag{4}$$

where $p(a_i)$ is the probability of symbol a_i. Entropy measures the randomness (or lack of predictability) in the data. Datasets with biased probabilities have a much lower entropy than regular iso-probable datasets. For example, a dataset with 500 a symbols and 500 b symbols will require 1 bit per symbol:

$$\begin{aligned} H(A) &= -\sum_i p(a_i) \log_2 p(a_i) \\ &= -\left(\frac{500}{1000} * \log_2 \frac{500}{1000} + \frac{500}{1000} * \log_2 \frac{500}{1000}\right) \\ &= -\left(0.5 * (-1) + 0.5 * (-1)\right) \\ &= 1 \end{aligned}$$

On the other hand, if there are 900 a symbols and 100 b symbols, we obtain a smaller entropy:

$$\begin{aligned} H(A) &= -\left(\frac{900}{1000} * \log_2 \frac{900}{1000} + \frac{100}{1000} * \log_2 \frac{100}{1000}\right) \\ &= -\left(0.9 * (-0.15) + 0.1 * (-3.32)\right) \\ &= 0.46 \end{aligned}$$

Entropy is not dependent on the order of the symbols, just on their probability. Entropy is the limit of compression that can be achieved given a set of symbols with associated

probability. In a stream of bits, the probability of a symbol is determined by the frequency of that symbol on the stream.

Context encoding uses previously decoded symbols to alter the probabilities of the next symbol. In some domains the information of what symbols have been encoded recently changes the likeliness of what is going to be the next symbol. For example, the letter l in English is more probable after an e than after an h. Through the modification of the probabilities, the entropy of the dataset is reduced and the file size becomes smaller. Context encoding is used in text compression as well as some image compression (PNG [86]). Extrapolating context encoding to geometric data compression is not feasible because the large number of different values makes the total number of possible contexts unmanageable. For example, a context of two symbols, where a symbol is an unsigned 32 bit integer, requires $2^{32} * 2^{32}$, 2^{64} memory positions. There is no computer with that much memory, and using context with more symbols increase the memory cost geometrically. The method inspired in context prediction for geometric data is geometric prediction.

Geometric prediction exploits coherence between neighboring values and predicts the value to be encoded as a function of previously processed values in the neighborhood. For example, in curve compression, a geometric predictor can fit a line to the two previous values. The prediction seldom is perfect; the computed value is not exactly the same as the real one. The residue is the difference to be added to the prediction in order to obtain the correct value. Geometric prediction thus transforms a set of values into a set of residues. Residues tend to be centered around zero and hence have a much more biased probability than the real values, improving the compression. A simple example can be seen in Figure 6. It depicts a simple function, a sinusoidal, quantized to 2000 different values. The entropy of the raw values is five times larger than the entropy of the geometric residuals. The prediction used in this example is line fitting.

Often compression is evaluated by compression ratio. Compression ratio is defined as the ratio between the uncompressed size and the compressed size. It is expressed as a factor. A data that needs 100 bits and is compressed to 40 bits has $100 : 40$, $10 : 4$ or $2.5 : 1$ compression ratio.

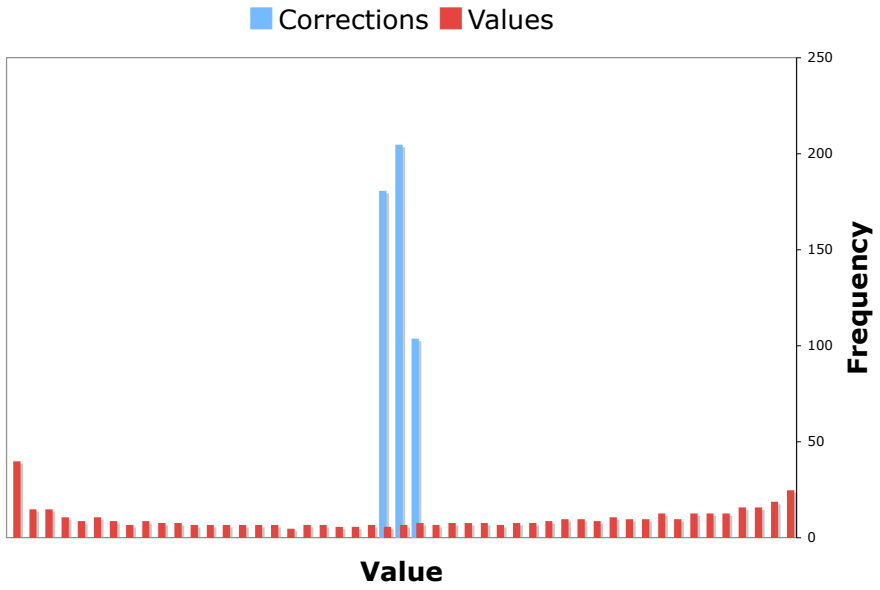


Figure 6: Histogram of values for a sinusoidal function and corrections of linear prediction applied to the function. The entropy of the values is 8.70 bits per value, while the entropy of the corrections is 1.53 bits per value.

It is important to distinguish lossy compression and lossless compression. Lossless compression preserves the original information. Lossy compression encodes an approximation of the input data. Lossy compression can usually achieve better compression ratios than lossless compression.

Quantization is the process that assigns discrete finite values to a continuous signal. In compression, quantization is the process that maps a set of input values to another set of input values, usually smaller set of integer numbers. Quantization represents the same data with fewer bits that suffice to identify a symbol from the new set.

2.3 Objectives

Increasing the compression ratio is the main objective of this research. Applications that use compression, while requiring as much compression as possible, also have other needs. Decompression may not be a trivial process, there are schemes that require sequential access to the compressed data. As such, random access to the compressed information is not viable. The memory needed for compression and decompression should be only a portion

of the dataset, and in some applications the data should be transmitted progressively. Progressive transmission implies that the information is transmitted as a rough approximation and is refined with each decoded set of bits. The decompressed data can be used before decompression has finished.

Let us consider briefly several applications. Given a regular grid and a selected IsoSetRange, the user might wish to explore neighboring isovalues. Our algorithm supports extension and refinement from the already processed IsoSetRange. Meshes and isosurfaces can be very large, and many times there is no need to decompress the whole mesh. Our approaches (hierarchical compression using Spectral predictors, for example) allow progressive decompression: first a low resolution model of the data is obtained, later it can be further refined if needed. Furthermore, if the refinement is only relevant in an area of the data, we propose to only refine that part. For example, in a low resolution map of Georgia we want to have a high detail of Atlanta without also decoding Athens. This also applies to the precision of the values. A quantization of few bits should be able to be extended to higher precision.

Our approaches have been designed with streaming in mind. Streaming is the property that the input data is processed only once, resulting in a memory buffer of a small portion of the data. In a real life feed of data, the option to do two passes on the data is not available; the data must be compressed when it is received. As a result, streaming programs have constraints on processing speed and memory usage.

CHAPTER III

GENERAL PRIOR ART

3.1 Lossless compression

In this thesis we compress data. The technique used to compress and encode the symbols after compression transformations is symbol compression. A widespread method for fast symbol compression is Huffman code, which was developed by Huffman in 1952 [47]. Huffman code is a technique that uses variable bit length encoding. The number of bits used to encode each symbol is dependent on the probability of such symbol. More frequent symbols have codes with shorter length. Less frequent symbols may have codes with lengths larger than the one they had on **raw** encoding, but the extra cost of encoding them is offset by the gains on the more frequent symbols. To compute the code of each symbol, the symbols are stored in a binary tree, and the code of a symbol is defined by the path to traverse the tree from the root up to the encoded symbol; 0 if you go left and 1 if you go right. The Huffman tree has the property that symbols with higher frequency are closer to the root.

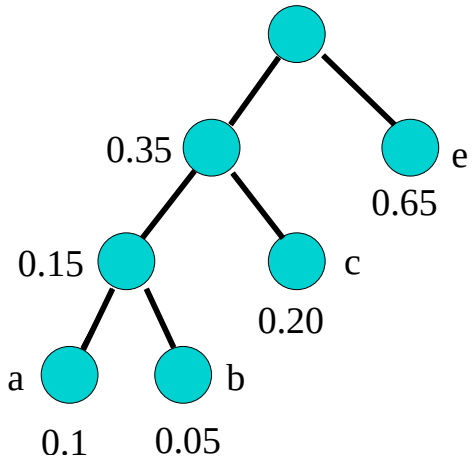


Figure 7: Huffman tree: each node contains the sum of probabilities of all its children.

Generating a Huffman tree is fast and simple. First all the symbols are sorted by their probability, and each one is a tree. Then we repeat the following process: the two trees with the smallest probability are merged. The root of the merged tree has the sum of the

probability of the two children. When all trees are merged, we have completed the Huffman tree.

Huffman coding compression is near the optimal for cases when the symbols have probabilities of the form $\frac{1}{2^i}$ for some i. Such probabilities are not common, hence the Huffman code has room for improvement. Huffman coding has been improved in many fashions, from using an n-*ary* tree (considered by Huffman in his original paper [47]) to using a constrained length Huffman tree and adaptively growing tree (as proposed by Vitter [103]). Huffman coding produces good results.

Arithmetic Encoding [107] addresses the shortfall of Huffman coding. One way to improve on Huffman efficiency is to group symbols together in blocks. The larger the blocks are, the better the symbol code length is. But for an alphabet of size k, if the blocks are of size n, the total number of blocks becomes k^n. Soon the Huffman tree of the set of symbols becomes too large to fit in memory. Arithmetic encoding is a technique that effectively encodes the whole string of symbols as a block. Arithmetic encoding is not trivial to implement, but it offers several features: for large enough string of symbols, the compression ratio is asymptotically the entropy, which is optimal. Arithmetic encoding can change the symbols and their probability at any time, taking advantage of cases when a symbol cannot occur, or when the probabilities are not know and are learned on the fly (adaptive model). All of this comes at the cost of more processing complexity, making arithmetic encoding slower than Huffman encoding. Huffman encoding consists of traversing the tree to determine the code, and writing the code to the output. Arithmetic encoding requires maintaining interval structures and dealing with precision overflows. On top of all, Huffman codes can be stored in a table of constant access, while arithmetic encoding requires computing for each symbol the corresponding set of bits to encode.

Let us imagine that we have a computer with infinite precision. We wish to encode the string *abc*, where the probabilities of the symbols are 0.5, 0.25 and 0.25 respectively. Arithmetic encoding divides the real segment between 0 and 1 according to the probabilities. For the first symbol, with the above probability table, any number between 0 and 0.5 encodes that the first symbol in the string is an a. A number between 0.5 and 0.75 encodes that

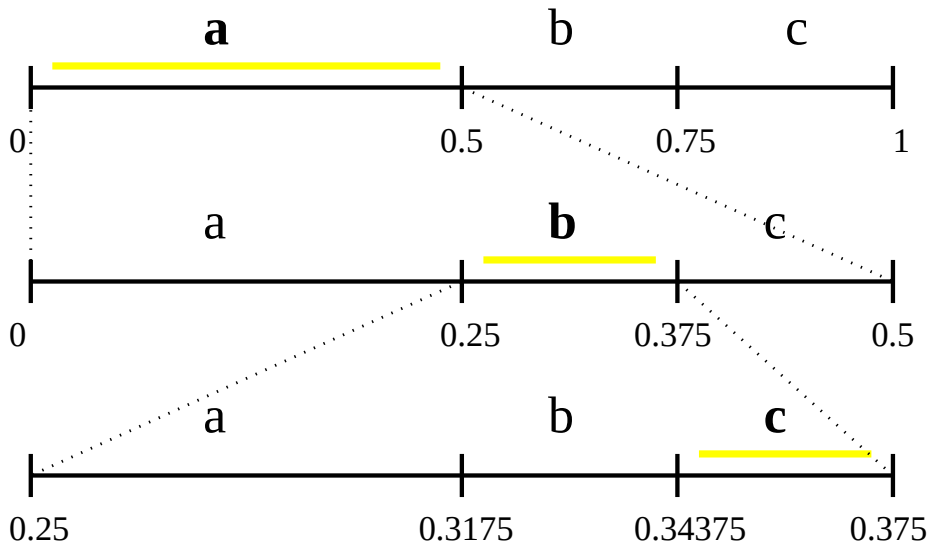


Figure 8: This figure depicts how the intervals are being divided with the arithmetic coding algorithm.

the first symbol in the string is b, and so on. For the second symbol in the string, the interval from encoding the first symbol is divided again according to the given probabilities. To encode ab, arithmetic encoder needs to encode a number between 0.25 and 0.375. To encode the whole string, a number between 0.34375 to 0.375 must be encoded. As the probabilities diminish and the string becomes longer, the intervals become smaller. This leads to a precision problem, which arithmetic coding implementations solve by writing to disk the bits encoding the common prefix of both ends of the interval. For example, for the 0.34375 to 0.375 interval, the 0.3 prefix can be encoded. We encode information to determine the 0.3, and scale back the interval to gain precision. After the encoding process, the compressed stream contains a number inside the smallest of the computed intervals. Separately from that number, the total number of encoded symbols (and thus the number of intervals) is encoded.

The corner stone of compression is redundancy. All compression methods represent their input information without its redundancy, thus making the representation of the information more compact. Redundancy appears in different fashions, and the more that is known about the input information (constraints, characteristics) the better the compression can be. A very simple way of removing redundancy is the run length encoding **RLE** compression

method. Considering the input information a string of symbols, for each run of consecutive identical symbols, RLE outputs two items: the length of the run and the symbol to be repeated. Consider this the input string of symbols:

```
WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWWBWWWWWWWWWWWWWW
```

The RLE output is:

```
12W1B12W3B24W1B14W
```

The number of bits used to encode the length of the run, a run of length 12 on our example, is a design decision of the RLE encoding. In our example, 5 bits can be used to encode the length. If several runs were of length smaller than 16, with a 5 bit encoding there would be a waste of 1 bit per run. On the other hand, if several runs were of length longer than 64, a RLE with a 5 bit length encoding would need to split a run into two smaller runs, thus increasing storage.

There is a large compression ratio using RLE in this example. When the input information does not contain a symbol repeated many times, RLE can actually produce an encoding that takes more bits than the raw encoding. Many ideas have been proposed that build on top of RLE, but RLE is still used nowadays due to the simplicity of implementation and speed of decoding.

RLE treats the symbols one by one. Often there is correlation between groups of symbols, not only individual symbols. Dictionary methods such as LempelZIP (LZ77/LZ78 [110], LZW [106]) create a repository of previously seen strings and assign a code to them. The repository has all the symbols and the string of symbols that have occurred in the dataset. As strings grow in length, the same code will represent more symbols thus achieving better compression. If a string is not seen anymore, new strings replace the old ones from the repository. The repository is called a dictionary. The symbols in the dictionary are encoded using Huffman coding. This technique is popular because it can be applied to any string of symbols and has good compression/decompression speeds.

The previous compression techniques do not apply well to floating point data. There are too many floating point numbers to be considered a set of symbols, and compressing

each byte on its own is not using the fact that the bytes represent floating point numbers. Isenburg et al [53] have proposed a method for the lossless compression of floating point data. Each floating point number is composed of a sign bit, a mantissa and an exponent. Isenburg's technique is geared towards geometric prediction; it encodes not the residual, but a set of bits indicating how to transform the predicted floating point value into the correct floating point value. Their technique stores the integer variation between the exponents, encodes the length of bits of the mantissa that are equal, and sends verbatim the mantissa bits where the two numbers differ. Both the residual in exponents and the length of bits for the mantissa are encoded using an arithmetic encoder. This approach uses only integer arithmetic and allows for fast, efficient and lossless compression of floating point data. Isenburg and Lindstrom [68] have extended their approach by removing the encoding of the exponent. Their new approach encoded the position of the first different bit from the predicted and the correct value, and the stream of bits from the correct value. Other approaches have been proposed that focus more on speed than compression [16], but for our purposes we follow Isenburg et al's scheme.

3.2 Lossy compression

All the compression schemes described so far represent the information in a compact form, but they represent the exact same information. The compression encoding process is reversible. Compression schemes with this characteristic are called lossless, because there is no loss of information. Lossy compression, which may help further increase compression ratio is not able to recover perfectly the input information, it recovers an approximation. By introducing errors, or not fully encoding all the details of the information, the compression ratio achieved by lossy programs is typically between 10 and 100 times more than that of a lossless scheme, comparing lossy JPEG image compression with lossless image compression. Not all types of information can be compressed with a loss. Binary programs, medical images are good examples where fidelity is required. In cases where the ultimate user is a human, such as images for visualization, videos, sound clips, triangle meshes, the

introduction of hardly noticeable errors may be acceptable. There is a hybrid approach between lossless and lossy compression, the progressive compression. Progressive compression encodes all the information in such a way that the encoded data from the start to any point provides an approximation of the input content. The whole compressed file can decode the input information without any loss.

Quantization is a common method to compress numeric values. It is by default a lossy method of compression, for its lossless version would not decrease the file size in most of the used input datasets. In its simplest form, a quantizer observes a single number and selects the nearest approximating value from a predetermined finite set of allowed numerical values. The number of bits required to perform this encoding depends on the number of allowed clusters of values. If the values in the original dataset are evenly distributed, uniform quantization is at its best. The error introduced by quantization will be uniformly spread, and all the clusters from quantization will have a similar area of values mapped to them. When the dataset has its values concentrated in a small region, uniform quantization allocates bits in areas of the interval that are not used, while assigning the concentrated values to a few clusters. As a result, there are values in the quantized version that are not used; no value from the original set maps to them.

A fundamental result of Shannon's rate distortion theory [96] is that better performance can always be achieved by coding vectors instead of scalars; the vectors are a block of the former scalars. Vector quantization maps each vector of values to a code. Supposing that there is correlation between the values, vector quantization takes advantage of this correlation, whereas single value quantization removes such correlation with the introduced error.

Vector data appears in images, where each pixel has three components, as well as in scientific datasets [23]. Quantizing such data in vector format is a complex process. Depending on the error metric, one quantization can be considered better than another; due to the high complexity of the data a common solution is to provide approximations. A popular technique to address this problem is k-means, also known as Lloyd's algorithm.

K-means [31, 74] is a common technique to cluster values and it is used frequently

to perform non-uniform quantization. K-means first select N representatives. All the values to be quantized are mapped to one of the representatives. For each cluster, a new representative is chosen. Usually it is the average of all the cluster members. The process is repeated again and again until the error is minimal enough or the time allocated to it has finished. K-means implements non-uniform quantization, allocating the bits of the quantized clusters to areas with more values. K-means favors minimizing the error of large number of points; if few points are far away from the populated clusters, those points will not be represented in the quantization. This results in less L^2 error (mean square error), but it is not admissible in the fields this thesis approaches. The use of K-means provides no control over the maximum error. A large error in one value could mean a change in topology for triangle meshes, or for isosets.

Lloyd's approach is not scalable to high dimensions without some extra care. When the number of dimensions increases, the task to associate a vector with a representative becomes time consuming. Gray et al [35] use a tree structure to hold the representative vectors. The tree structure defines the code each representative is assigned and facilitates the process of matching a vector with its representative. Cosman et al [24] discuss the problem of quantization with respect to image processing. Both balanced and unbalanced structures are reviewed, with a preference for unbalanced structures where more common representatives are assigned a shorter code. When applying quantization to images, it is possible to have a different criterion than the minimization of L^2. Cosman et al show quantization applied to enhance the visualization of an image by quantizing the pixels with the purpose of equalizing the colors.

3.3 Predictive Compression

Predictive Compression is the family of techniques that predict the next value in the dataset and encode the residual or a symbol to allow the decoder to obtain the correct value from the prediction. Predictive Compression can be applied with several other methods, such as frequency decomposition [104], Principal Component Analysis [57], or just by itself (PNG [86]). The previous techniques transform the information (frequency transforms, PCA transforms)

and use predictive compression on the transformed set.

The predictors used must only be computed with already processed values, otherwise the decoder cannot reproduce the same prediction as the encoder.

Predictors have a *mask*, the neighborhood of points from which the predictor is computed. Larger masks enable predictors to capture more subtleties in the data, but are much more sensitive to noise and quantization error.

The Motion Pictures Experts Group devised a scheme to compress a sequence of images through time, the MPEG algorithm. The MPEG standard has several revisions. MPEG4/AVC is used as standard in the industry nowadays, and it uses prediction from previous frames to encode the new frame. Still images are compressed with JPEG; they are divided into blocks, but the blocks are predicted from the previous frame.

Predictive Compression techniques are favored for the ease of implementation, the low computation cost associated with a predictor, and the control the application has over the predictor's behavior.

3.4 Signal Based Compression

Fourier theory states that all signals can be expressed as a weighted sum of sinusoidal functions of different phase and frequency [14]. This process, known as spectral decomposition, has fathered an important area of data compression. For all the datasets that represent a function, such as audio, images, video (up to a certain extent) and geometry, spectral decomposition transforms the data from a set of values to a set of frequencies with weight. The process is invertible; the transformation can be done so there is no error introduced. There would be no error if the numbers had infinite precision; therefore a careful computation method has to be applied.

The data we work on often has information dependent on frequencies. Humans perceive high frequencies as details and low frequencies as main features, although this is a generalization. One major benefit of spectral decomposition is that the different components of the information can be treated differently. It is possible to completely remove all the detail, and achieve compression through its omission. It is possible to quantize different frequency

streams using different parameters, based on their relevance.

Spectral decomposition is heavily used in image compression. JPEG [104] is a standard that uses spectral decomposition. JPEG processes the input image in blocks of 8×8 pixels. For each block, JPEG computes its spectral decomposition and quantizes the coefficients. From a human perception point of view, the degradation in quality is acceptable, and the technique achieves good compression rates; JPEG is the most used image format nowadays. It is possible to control the level of compression by deciding how many coefficients to discard. JPEG has a provision to encode the information not by block, but by level. Hence, the image is transmitted in a progressive fashion.

JPEG has two main drawbacks. It is not able to perform lossless compression, and due to its treatment of the input in blocks, it is possible to assert seams in the compression (discontinuities in color and edges between two 8×8 blocks), artifacts due to the different ways two contiguous blocks were encoded. Images for which it is important to preserve edges, such as those that contain text, are not well suited for the JPEG algorithm.

Wavelets are a very well studied compression technique [7, 19, 39]. The wavelet approach consists of filtering the data to obtain a version of the data with high frequencies removed. The high frequencies are encoded and the remaining data is again filtered to remove frequencies. Wavelets have been applied to a myriad of different types of data. From one dimensional datasets to several dimensions, passing through meshes and animations. Most wavelet techniques work on one dimensional signals; to apply them to data of higher dimensionality it is only required to process one dimension at a time. For example first apply wavelets in the X axis and then apply the same wavelet technique to the result in the Y axis.

Wavelets apply a filter to the signal that produces two outputs, a low frequency part of the signal and a high frequency part. The high frequency part of the signal is the detail of the signal, and there is few coherence left in it. The low frequency part of the signal is sent to the input of the filter again. Common wavelet filters are the Haar and Cubic Filters. The Haar filter sets the low frequency part of the signal to be the average between each pair of values, and the detail to be the difference between the average and one of the

values in the pair. The Cubic filter fits a cubic to four consecutive values to obtain the low frequency component, and sets the detail to be the deviation between the values and the low frequency component. The difference between wavelet filters determine how much coherence the filter expects to find in the data, and how sensitive to noise the filter is. Haar wavelets use a small filter footprint, and they are able to work with data containing noise. Cubic wavelets have a larger footprint than Haar wavelets; Cubic wavelets can extract more coherence in the low frequency decomposition, but are more susceptible to noise than Haar wavelets. Cubic wavelets compute the detail coefficients as the deviation of a locally fitted cubic function. Signals with noise influence the fitted cubic function; the noise appears in the low frequency, making the high frequency part of the signal have larger magnitude than Haar wavelets, where only two points contribute with noise instead of four.

The main focus of wavelet techniques is on progressive decomposition of data; lossless compression entails the encoding of all levels of details. Commonly specialized lossless compression techniques outperform wavelets.

Wavelets, by the very nature of the filter approach, treat the data in a progressive fashion. If the wavelet decoding process were to be stopped at some point, the data decompressed would be a low frequency representation of the final signal, thus making Wavelets an incremental decompression technique. While it is possible to store the detail part of the signal using symbol encoding, wavelets are often encoded with a zero tree, to take advantage of the common prefixes in the wavelet detail streams. A zero tree organizes the bits to be encoded by relevance, or by the precision they add. All the bits of the same relevance are encoded together, as if they were in a layer. Said et al [92] developed SPIHT, a technique to do zero-tree encoding of wavelet coefficients with hierarchical trees.

CHAPTER IV

REGULAR GRIDS

This chapter describes our contributions to the regular grid compression area. In Section 4.1 we describe the problem of grid compression. We survey the literature of regular grid compression in Section 4.2. Our compression methods are based on prediction, the prediction used is determined by the order in which the values of the grid are accessed. We discuss several grid traversals in Section 4.3. We propose an array of different predictors in Section 4.4. We discuss our results in Section 4.6.

4.1 Introduction

Numerous engineering, biomedical, and other scientific applications produce extremely large datasets through numeric simulations or physical data acquisition. In a large proportion of the cases, the data represents one or more scalar fields sampled over regular grids in dimension three, four, or higher. For example a typical 3D simulation produces values on a regular grid of $2,048^3$ samples [83]. In 4D a typical combustion simulation generated using a High-Performance Parallel Processing Cluster may include 1,000 time slices, each representing a regular sampling of a cube at a resolution of 512^3 [76]. Another example may be fluid dynamics data, used in our tests [23] (see Fig. 9).

The acquisition or computation of most scientific datasets [23], high dynamic range images [62], or videos usually requires a significant amount of effort and computing resources. Yet, their exploitation is often hindered by the mismatch between the size of the files in which they are stored and the available bandwidth for downloading or visualizing them. Although the loss of accuracy resulting from a controlled quantization or lossy compression may be acceptable for visualization purposes, lossless compression of integer or floating point values is required in many settings to guarantee the integrity of the data, especially if it is to be used to save state in "restart dumps" to allow resuming an interrupted simulation [23]. Furthermore, it is often desired that the data be compressed as it is created,

streamed, and decompressed using a small memory footprint.

Whereas traditional image compression techniques are capable of lossless compression [86, 93, 105], they were developed for the media industry which usually deals with low range data and tolerates trading some accuracy for increased compression. In contrast, this thesis focuses on the lossless compression of high range datasets represented for example as 32-bit integers or floating point numbers.

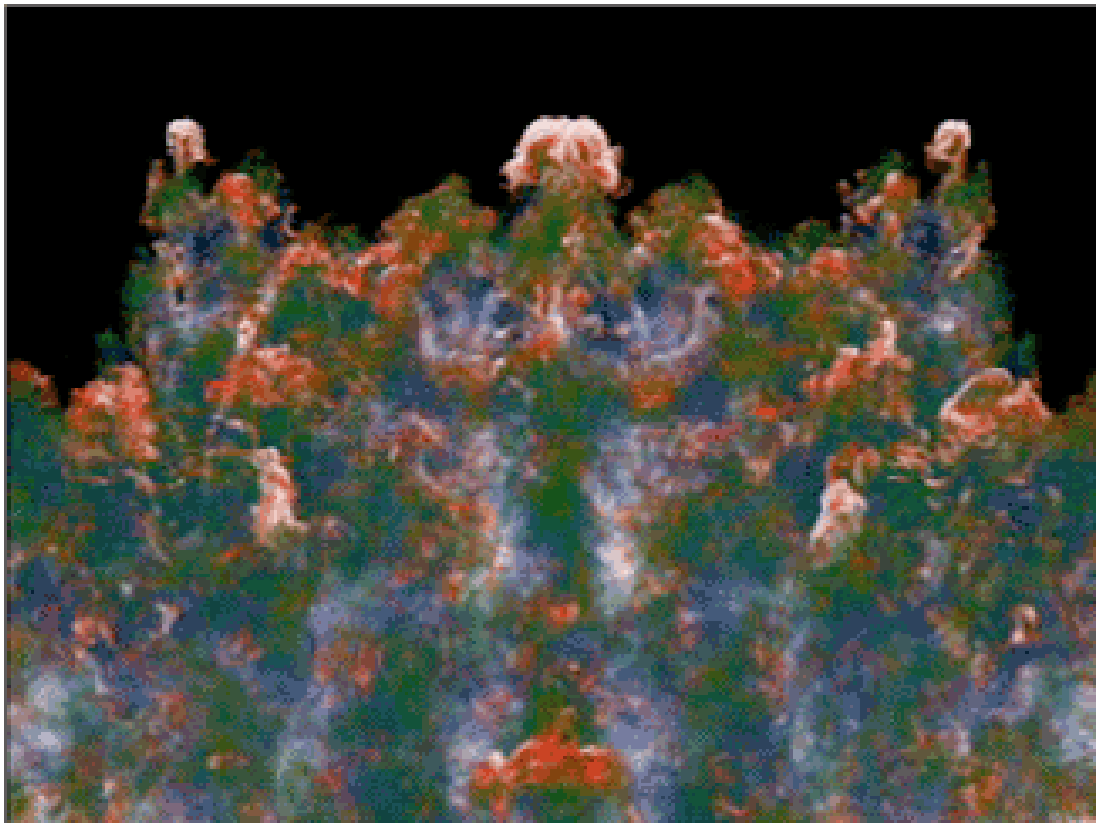


Figure 9: A 4D data set from a simulation of two fluids interacting. This image is courtesy of Lawrence Livermore National Laboratory.

The compression and streaming approaches exposed here follow the predictive compression paradigm: compute the prediction $p_{i,j}$ of the scalar value $f_{i,j}$ at each sample (i, j) from values of previously processed samples in the neighborhood $N_{i,j}$; compress the corrections, $c_{i,j} = f_{i,j} - p_{i,j}$, using lossless entropy coding or a custom format designed for compactly encoding differences between nearby floating points; and stream them. The paradigm leads to simplicity of implementation, small memory footprint, and excellent compression.

4.2 Prior art in compressing Regular Grids

Several compression techniques have been proposed for lower dimensional gridded data. These include image and video compression techniques, as offered by the MPEG-4 standard [32] and various volume compression approaches [9, 30].

A variety of methods to compress 4D volumes have been proposed in recent years. These include wavelets [40] (discussed in Section 3.4), discrete cosine transform (DCT) [72] and run length encoding (RLE) [5]. The wavelet approach uses an interpolating predictor, which, according to informal experiments, produces 50% smaller residuals than an extrapolating predictor. On the other hand, the wavelet's hierarchical approach requires more space and processing power than our extrapolating predictor. When local temporary storage is an issue, wavelet approaches may break the dataset in smaller chunks and compress each one independently. The proposed approaches do not require such splitting.

Fowler and Yagel [30] proposed a similar approach to ours. They also used previously decoded samples to predict a new value in 3D volumes. They used the three nearest visited neighbors, and compute optimal coefficients for their predictor. Ma et al [73] computed an octree for each frame, and encoded the differences between octrees. They also compared uniform and non-uniform (or adaptive) quantization of the data before compression.

Others quantized the corrections or the wavelets coefficients, e.g. Bajaj et al. [9]. On the contrary, we quantize the dataset and then perform lossless encoding. As a result, the maximum error produced by our approach is given by the quantization error, in case we quantize the data.

Out-of-core methods for simplifying [67] and compressing [42, 49] 3D polygonal meshes have recently been proposed. Out-of-core methods for compressing 3D volumetric data sets have also been proposed [22]. Chiueh et al's approach segments the volume into different chunks, transforms each chunk separately using a Fourier transform, and encodes the transformed result.

The standard approach to lossless compression of such data is based on *predictive coding* (parallelogram predictor [68, 101], polynomial fitting prediction [28], or prediction based on previously seen contexts [85]), and several prediction schemes for structured datasets have

been proposed [29, 60, 81, 105]. These prior schemes work best when the traversal over the data is simple, e.g. scanline order, so that each sample can be predicted from a single spatial configuration (stencil) of nearby, previously coded samples. When more general traversals are desired or when a nontrivial subset of samples is requested, the configuration of nearby known samples is often irregular and changing, which normally requires falling back on simpler predictors involving fewer samples. We propose techniques for the scanline traversal and for more complex traversals.

4.3 Data Traversals

In a computer a regular grid is often represented as a multi-dimensional array, a matrix. This matrix is stored *by-rows* on memory, and on disk. To design an algorithm that processes all nodes in the grid, it is necessary to devise a traversal, an order in which all nodes will be accessed. To increase efficiency, a node should be accessed as few times as possible. Furthermore, disk access and cache coherence are important factors to take into account. Hence, it is preferable to access nodes in an order that reduces page faults.

4.3.1 Scanline Traversal

As pointed out above, it is advantageous to access the grid in the same order it is stored in memory. Such traversal is called the *scanline* traversal.

Listing 4.1: Code for scanline traversal of a 3D regular grid.

```
for(z=0; z<MAX_Z; z++}
    for(y=0; y<MAX_Y; y++}
        for(x = 0; x < MAX_X; x++)
        {
            // process node value here
        }
```

Our techniques use predictive compression. To process the value at each node a set of neighbors has to be available to perform the prediction. The amount of memory to do this without reading a value twice is the footprint. The footprint is dependent on both the

traversal and the predictor. Also, it is important to be able to do streaming. To perform streaming, all values should be read from disk or written to disk only once. This requires to hold a value in the footprint memory until it will not be used any longer. For this reason, the footprint in scanline traversals is one or more slices of the data. For example, in a 512^3 regular grid, which takes a total of 512 Mbytes, the footprint in memory for a predictor that uses the previous slice is 1 Mbyte. Scanline traversal can be extended easily to grids of any size and dimension.

In the case that a user wants to obtain an approximation of the encoded data, for example when the user wants to determine whether the data is worth examining, it is required that data compressed using a scanline traversal is decompressed in its totality. The last value of the approximation might well be the last value of the data, which depends on nearby previously seen values, and the dependence permeates until the start of the data. Hence, all points have to be decoded even when only an approximation, a subset of the data, is needed.

4.3.2 Hierarchical Traversal

To allow partial decompression of data to be used as approximations, hierarchical traversal orders the values of the datasets in such a way that the first set of values is an approximation of the whole dataset, and the subsequent set of values is a refinement to the current approximation. This ordering permits to have access to the data in a progressive manner, and allows for complex predictions to be applied. This special ordering makes keeping a footprint a complex task.

As with the scanline case, the footprint is very dependent on the prediction used. In our experiments, we have computed a hierarchical traversal (see Figure 10) that requires $2 * lenght$ of the dataset as footprint, and we have also worked with a simple hierarchical traversal that requires half the input data as footprint. There is a tradeoff between simplicity and extension to higher dimensions, and memory requirements. Increased complexity makes extending hierarchical schemes to 3D, 4D and beyond a hard task.

If the dataset does not have size of the form 2^k+1, then it is common to pad the dataset,

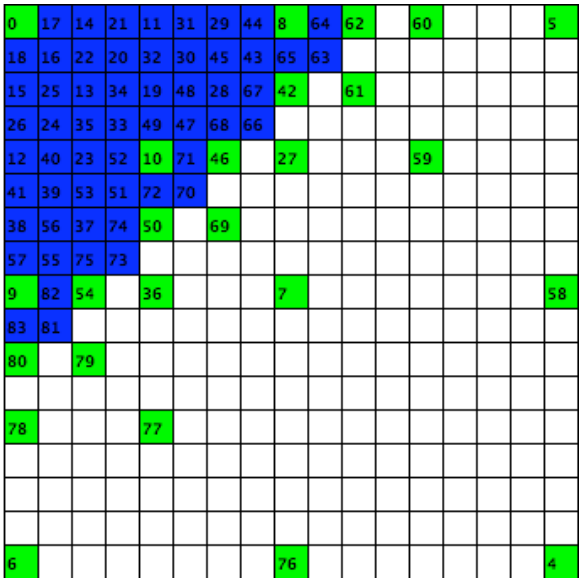


Figure 10: Trace of our hierarchical traversal, which requires only a footprint of twice the length of the dataset. Green are the points currently in the footprint, and blue the already processed points. The numbers indicate the order in which the value was processed.

even if the padding will not be used. Hierarchical schemes work better when the data size is of the form $2^k + 1$.

There are many choices of different hierarchical traversals. For the scope of this thesis, we focus on the hierarchical traversals that order nodes based on their coordinates. Any number can be expressed as a product of primes. The *level* of a coordinate is the power of 2 in its factorization. A coordinate of 24, which equals $2^3 * 3$, has level 3, for example. The level of a node is the minimum of the level of all its coordinates.

We traverse all the points on the same level in a scanline fashion, as described in Figure 37.

4.4 Regular Grid Predictors

Predictors use information from previously processed nearby points to guess what the next value will be. For cases where there is little correlation between values, the simplest prediction that can be used is to assume that the new value is the same as the previous one. This is called constant prediction, and as simple as it is, it is the best prediction for datasets where the noise is superimposed over the signal.

To improve the constant prediction one can use the linear prediction. In the 1D case, if we have values A, B, C, and we know B and C as in Figure 11(a), we can predict A as $2B - C$. This is the same as fitting a line through B and C and evaluating it at point A. Hence the residue is $A + C - 2B$.

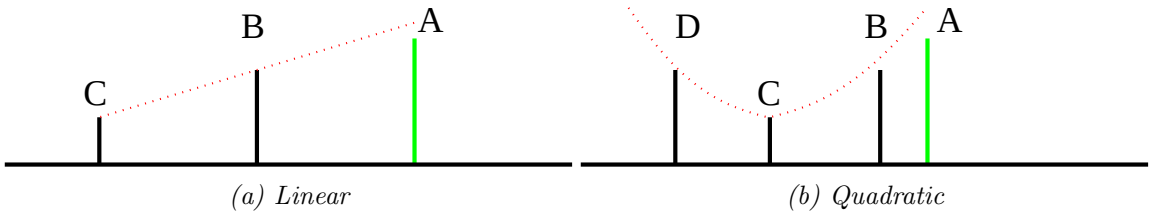


Figure 11: Example of basic extrapolation predictors in 1D. (a) linear extrapolation, (b) quadratic extrapolation.

The prediction can be extended to using three previous points. We add point D, and we fit a parabola on D, C, B and evaluate it at point A. Hence the residue is $A - D + 3C - 3B$. This is illustrated in Figure 11(b).

To predict A in (A, B, C, D) we assume that all four points lie on a quadratic function $f(t)$ so that $f(-1) = B$, $f(-2) = C$, $f(-3) = D$. The function f has the generic form of $f(t) = at^2 + bt + c$. Solving for the coefficient c yields the prediction for A.

To compute a prediction using n values, we create a polynomial of degree $n - 1$, and set up n equations, one for each value we have. With this, we can find the coefficients of the polynomial, and solve for the value we want to predict. Seldom this is done with more than four or five values in one dimension. The reason is that datasets are samples of the composition of functions of high degree, with added noise. A prediction with more values, which fits a high degree polynomial, will be polluted by noise from all the values, and will not be as accurate as a prediction with fewer values. For example, let's assume the function to be predicted is a cubic. Linear prediction uses the two previously seen values to fit a line. Quadratic prediction uses three previously seen values to fit a parabola. Near the origin, the previously processed points of a cubic define a positive parabola. The error of such parabola is greater than that made by the linear interpolation of the two nearby previously seen points. Ideally, the polynomial used should have the same degree as the

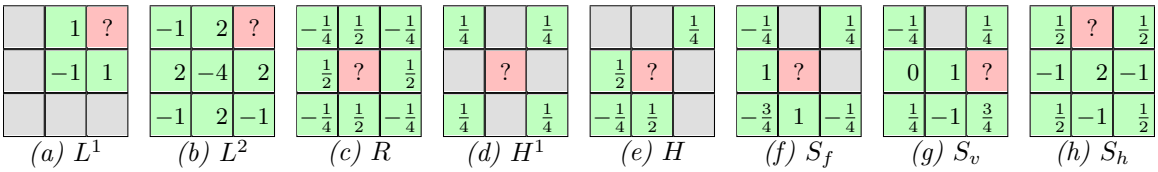


Figure 12: Weights for several spectral predictors used in our experiments: (a) Lorenzo, (b) bi-Lorenzian, (c) radial, (d) bilinear, (e) hybrid bilinear and radial, (f–h) full spectral.

implicit sampled function, if there is any.

So far, in the two previous examples, the value to be predicted was outside the convex hull defined by all the known values. We refer to this kind of predictors as extrapolating predictors. In the case the point to be predicted is inside the convex hull of all the known points, the predictor is an interpolating predictor.

4.4.1 Lorenzo Predictor, L^1

When applied to a sample v, the Lorenzo predictor estimates the scalar value $F(v)$ at v from its immediate neighbors that have already been processed. Both the compressor and decompressor visit the data in scanline order. For simplicity of notation, we use the local coordinate system where sample v has coordinates $\{1\}^n = (1, 1, \ldots, 1)$ and its previously visited neighbors are those samples with coordinates in $\mathbb{Z}_2^n = \{0, 1\}^n$. The value of the scalar field $F(v)$ is estimated from the field values at the other previously recovered vertices $U = \mathbb{Z}_2^n - \{v\}$ of the n-dimensional unit cube using the following formula:

$$E(v) = \sum_{u \in U} (-1)^{c_0(u)+1} F(u) \tag{5}$$

where $E(v)$ is the prediction of F at v and $c_0(u)$ denotes the number of coordinates of u that equal zero. Note that $c_0(u)$ may also be expressed as $c_0(u) = n - c_1(u) = n - u \cdot v$, where n is the dimension, $c_1(u)$ is the number of coordinates in u that equal one, and $u \cdot v$ is the dot product of u and v.

Note that in this formulation (Fig. 13), the immediate neighbors of the predicted vertex v have weight $+1$. Second degree neighbors (i.e., those which can be reached from v by traversing two edges of the cube) have weight -1, third degree neighbors have weight $+1$, and so on.

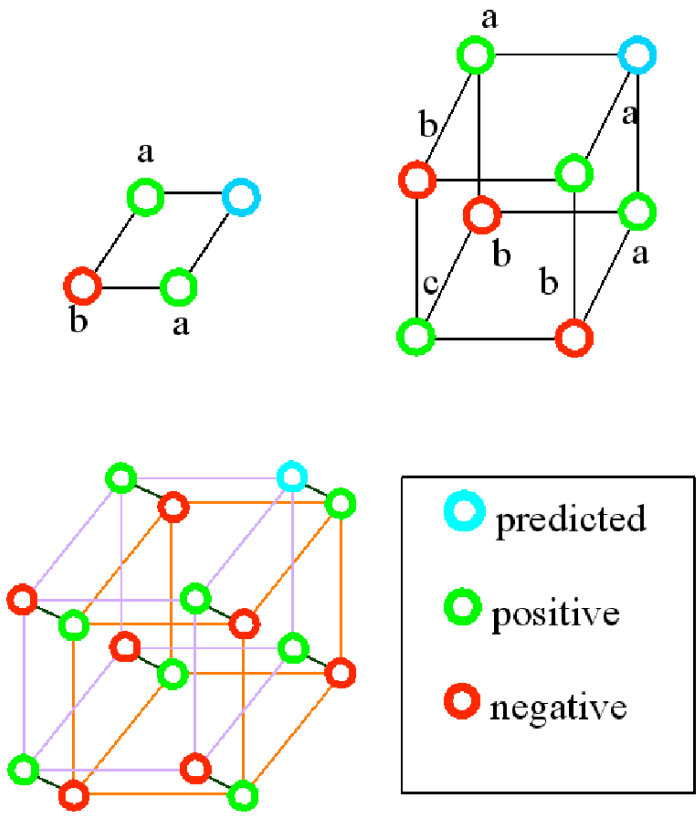


Figure 13: Lorenzo Predictors in 2D, 3D and 4D. In the 2D case (top left), the new value is predicted from its neighbors using the parallelogram rule (add the scalar field values at the two 'a' vertices and subtract the value at the 'b' vertex). In the 3D case, we add the values of the 'a' corners, subtract the values of 'b' corners, and add the values of the 'c' corner. In the 4D case, we add the values at the first and third degree neighbors and subtract the sum of the values at the second and fourth degree neighbors.

4.4.2 Prediction for polynomials

The mathematical derivation presented here is a contribution by Professor Andzrej Szymczak from the Georgia Institute of Technology.

The estimated values computed by the Lorenzo predictor in n dimensions are exact for all scalar functions that are polynomials of degree $n-1$. As a proof, assume that P is a polynomial of degree m in n variables, with $m < n$, and consider the following theorem and its corollary:

Theorem 1. *For a given monomial M in n variables of degree $m < n$, the sum of signed values $(-1)^{c_1(u)}M(u)$ over all the vertices u of the unit cube is zero.*

More formally, let $u = (x_1, \ldots, x_n)$. A monomial $M(u)$ has the form: $x_1^{p_1} \cdots x_{k-1}^{p_{k-1}} x_k^{p_k} x_{k+1}^{p_{k+1}} \cdots x_n^{p_n}$, with $\forall i\, p_i \geq 0$ and $\sum_i p_i = m$. The theorem states that $\sum_{u \in \mathbb{Z}_2^n} (-1)^{c_1(u)} M(u) = 0$.

Proof. There are n variables, but the sum $\sum_i p_i = m$ of the powers of the variables listed in M is less than n. Therefore at least one variable is not listed in M. Assume without loss of generality that the k^{th} variable is not listed. Consequently, the value of M is independent of that variable and thus $M(x_1, \ldots, x_{k-1}, 0, x_{k+1}, \ldots, x_n) = M(x_1, \ldots, x_{k-1}, 1, x_{k+1}, \ldots, x_n)$. Note that $(-1)^{x_1+\cdots+0+\cdots+x_n} = -(-1)^{x_1+\cdots+1+\cdots+x_n}$. Therefore, the vertices of the cube can be paired so that the values of $(-1)^{c_1(u)} M(u)$ on the two vertices of any pair are either both zero or have the same magnitude but opposite signs. Thus, the sum of the signed values is zero. □

This result may be easily extended to polynomials as follows:

Corollary 1. *For a given polynomial P in n variables of degree $m < n$, the sum of the signed values $(-1)^{c_1(u)} P(u)$ over all the vertices u of the unit cube is zero.*

Proof. $P = \sum_i M_i$ is the sum of monomials of degree $m < n$. We can permute the two summations:

$$\begin{aligned} \sum_{u \in \mathbb{Z}_2^n} (-1)^{c_1(u)} P(u) &= \sum_{u \in \mathbb{Z}_2^n} (-1)^{c_1(u)} \sum_i M_i(u) \\ &= \sum_i \sum_{u \in \mathbb{Z}_2^n} (-1)^{c_1(u)} M_i(u) = \sum_i 0 = 0 \end{aligned}$$

which proves the corollary. □

The corollary implies that

$$(-1)^n P(1, \ldots, 1) = -\sum_{u \in U} (-1)^{c_1(u)} P(u)$$

and hence

$$P(v) = (-1)^{n+1} \sum_{u \in U} (-1)^{c_1(u)} P(u) = \sum_{u \in U} (-1)^{c_0(u)+1} P(u)$$

As a consequence, in two dimensions the Lorenzo predictor is a linear predictor, and can exactly reconstruct portions of the scalar field that behave as a linear function.

$$F(x, y) = ax + by + c$$

In $\mathbb{R}^3$, the same simple Lorenzo predictor can reconstruct quadratic functions:

$$F(x, y, z) = \\ ax^2 + by^2 + cz^2 + dxy + exz + fyz + gx + hy + jz + k$$

In $\mathbb{R}^4$, the predictor extends its reconstruction power to all cubic polynomials, which are linear combinations of 35 possible monomials of degree of 3 or less in 4 variables. Even though the 15 values at neighboring grid points that the Lorenzo predictor uses do not provide enough information to compute all 35 coefficients of the polynomial, they uniquely determine its value at the corner v.

The Lorenzo predictor is of highest possible order among all predictors that estimate the value of a scalar field at one corner of a cube from the values at the other corners. In other words, the Lorenzo predictor is of optimal order for this setting, and no other predictor can correctly estimate all polynomials of degree n or higher.

As a justification, consider the monomial $x_1 x_2 \cdots x_n$ (the product of all coordinates). The product is zero on all vertices of the unit cube except one. So, a predictor would not be able to differentiate this monomial from the zero polynomial. Hence, the values of the scalar field at the $2^n - 1$ corners of an n-dimensional cube are not sufficient to recover the value of an n^{th} degree polynomial at the 2^{nth} corner.

Note that simpler predictors exist that correctly predict all polynomials of degree $m < n$. For example, one may use the n samples that precede v on a scanline. However, such lower-dimensional anisotropic predictors are much less effective since they fail to exploit data coherence in all dimensions. The Lorenzo predictor is the simplest isotropic predictor that can recover correctly all polynomials of degree less than n.

4.4.3 Extrapolating Bi-Lorenzo Predictor, L^2

−1	2	?
2	−4	2
−1	2	−1

Figure 14: Weights for the bi-Lorenzian predictor

It is natural to ask whether the Lorenzo predictor can be extended to higher-order polynomials that have vanishing higher-order derivatives. First, we can express the Lorenzo predictor as finite differences. Let f be a one-dimensional function regularly sampled at $\{\ldots, f_{i-1}, f_i, f_{i+1}, \ldots\}$, and let Δ^x be the finite difference

$$\Delta_i^x = f_i - f_{i-1} \tag{6}$$

That is, Δ^x is an approximation of the differential $\frac{\partial f}{\partial x}\, dx$. Setting $\Delta_i^x = 0$, solving for f_i, and substituting L_i^1 for f_i, we have as 1D Lorenzo predictor

$$L_i^1 = f_{i-1} \tag{7}$$

The Lorenzo predictor extends to 2D via composition of derivatives:

$$\Delta_{i,j}^{xy} = \Delta_{i,j}^x - \Delta_{i,j-1}^x = f_{i-1,j-1} - f_{i,j-1} - f_{i-1,j} + f_{i,j} \tag{8}$$

As the sampling rate of f increases, Δ^{xy} approaches $\frac{\partial^2 f}{\partial x \partial y}\, dx\, dy$ in the limit. Setting $\Delta_{i,j}^{xy} = 0$, we can now express the 2D Lorenzo predictor as

$$L_{i,j}^1 = f_{i,j-1} + f_{i-1,j} - f_{i-1,j-1} \tag{9}$$

Thus, in the limit, L^1 correctly predicts all continuous functions f with $\frac{\partial^2 f}{\partial x \partial y} = 0$. In the discrete setting, L^1 recovers linear polynomials, or equivalently bilinear polynomials without highest order term xy.

We can extend the predictor by taking finite differences once more and obtain

$$\begin{aligned} \Delta_{i,j}^{xxyy} &= \Delta_{i,j}^{xy} - \Delta_{i+1,j}^{xy} - \Delta_{i,j+1}^{xy} + \Delta_{i+1,j+1}^{xy} \\ &= 2f_{i,j-1} + 2f_{i-1,j} + 2f_{i+1,j} + 2f_{i,j+1} - 4f_{i,j} \\ &\quad - f_{i-1,j-1} - f_{i+1,j-1} - f_{i-1,j+1} - f_{i+1,j+1} \end{aligned} \tag{10}$$

where we define Δ^{xxyy} using central differences. Setting $\Delta_{i,j}^{xxyy} = 0$ and solving for $f_{i+1,j+1}$ we obtain the *bi-Lorenzian* predictor

$$L_{i+1,j+1}^2 = 2f_{i,j-1} + 2f_{i-1,j} + 2f_{i+1,j} + 2f_{i,j+1} - 4f_{i,j} - f_{i-1,j-1} - f_{i+1,j-1} - f_{i-1,j+1} \tag{11}$$

In the limit, L^2 reproduces functions f with $\frac{\partial^4 f}{\partial x^2 \partial y^2} = 0$, and in the discrete setting interpolates biquadratic polynomials without highest order term x^2y^2. Whereas Δ^{xxyy} relates to

Δ^{xy} as Δ^{xy} relates to f, L^2 is usually not the successive application of L^1, i.e. in general $L^2_{i,j} \neq L^1_{i,j-1} + L^1_{i-1,j} - L^1_{i-1,j-1}$. Instead, L^2 may be derived by setting to zero the L^1 correction of the L^1 corrections at (i, j). The L^2 weights are shown in Figure 12(b).

The L^1 predictor has been widely used in the image and geometry compression communities [86, 101, 105]. We are, however, not aware of its extension L^2 having been used for compression of 2D and higher-dimensional data.

4.4.4 Interpolating Predictor, R

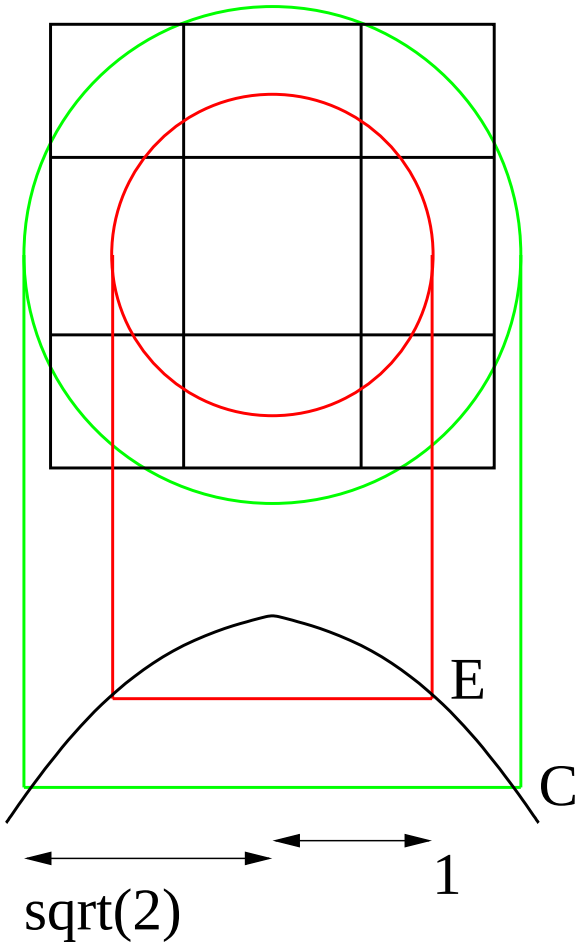


Figure 15: Rationale for the Radial Predictor. The red circle represents the average of points at distance 1 from the center, and the green circle represents the points at distance $\sqrt{2}$. The radial predictor is computed as a linear extrapolation from both averages, $2E - C$.

In the previous section we presented an extrapolating predictor, L^2, for a corner $f_{i+1,j+1}$ of a 3×3 neighborhood of samples. This predictor arose from the constraint $\Delta^{xxyy}_{i,j} = 0$,

a central difference evaluated at the *center* sample of this neighborhood. A higher quality prediction is obtained by solving Equation 10 for the function value at the center sample $f_{i,j}$, which results in the *radial* interpolating predictor

$$R_{i,j} = \frac{1}{4}\big(2(f_{i,j-1} + f_{i-1,j} + f_{i+1,j} + f_{i,j+1}) - (f_{i-1,j-1} + f_{i+1,j-1} + f_{i-1,j+1} + f_{i+1,j+1})\big) \quad (12)$$

We use the term "radial" to describe this predictor because its weights are radially dependent on the distance to neighboring samples (Figure 15). The predicted value $R_{i,j}$ is $2E-C$ where E is the average of the four edge neighbors $\{f_{i\pm1,j}, f_{i,j\pm1}\}$ and C is the average of the four corner neighbors $\{f_{i\pm1,j\pm1}\}$. $R_{i,j}$ also equals the mean of the four possible L^1 predictions of $f_{i,j}$.

R has the same predictive power as L^2, i.e. it reproduces biquadratics with no x^2y^2 term, but typically yields better predictions due to the symmetric configuration of its neighborhood samples. Using Taylor expansion of f one can show that the prediction error of L^2 is $\frac{\partial^4 f}{\partial x^2 \partial y^2}$ (plus higher order terms), while the prediction error for R is only one fourth as large. Note that to use R, we either must know all eight surrounding neighbors or must estimate them via alternative predictors, which in fact amounts to a different predictor.

4.4.5 Spectral Predictor, S

The spectral predictor S generalizes L^2 and R to all possible configurations of 0 to 8 known samples in a 3×3 neighborhood in which the sample to be predicted is arbitrarily located.

Many image compression techniques rely on transforming the image data to the frequency domain, either via discrete wavelet [19] or cosine transforms [104], where high frequencies corresponding to noise or "unimportant" features are attenuated or discarded altogether. We extend these ideas to design as-smooth-as-possible interpolants for irregular sample configurations. We seek to eliminate or, when not possible, to minimize high frequency responses. The resulting predictors and their set of weights are straightforward to use in a compression scheme, and can be stored in a lookup table, which for any given mask of known and unknown values and location of predicted sample returns a set of coefficients.

We build upon the work by Isenburg et al. [50], who use the Fourier transform to predict the geometry of n-sided polygons to be "as regular as possible" given $m < n$ known

vertices. They express the vertex coordinates of the polygon in the complex plane, apply the discrete Fourier transform (DFT) to this n-vector of consecutive vertex coordinates, set the highest $n - m$ frequencies to zero, and compute the inverse transform to obtain the complex coordinates of the predicted vertices. Because the Fourier transform is linear, the unknown vertices can be expressed in terms of a linear combination of the known vertices. By working out the mathematics of the forward and inverse Fourier transforms, one can a priori establish a set of weights to be used for a given configuration (m, n) of known and unknown number of vertices (i.e. the weights are not dependent on the geometry of the known samples). Because Fourier frequencies come in pairs, this approach works well when m is odd as then the resulting weights are guaranteed to be real. One can show that the discrete cosine transform (DCT) can instead be used when m is even. Lifting the DFT to higher dimensions, Isenburg et al. further showed that the L^1 predictor is in the spectral sense the optimal predictor (i.e. smoothest interpolant) for hypercube-like neighborhoods with one unknown sample.

4.4.5.1 Derivation

The mathematical derivation of the Spectral predictors was developed by Dr. Peter Lindstrom at the Lawrence Livermore National Laboratory. We begin by extending the general approach of Isenburg et al. to 3×3 neighborhoods to re-derive the bi-Lorenzian and radial predictors and show that they are optimal. We will make use of the two-dimensional discrete cosine basis

$$\{u(x)u(y), s(x)u(y), u(x)s(y), s(x)s(y), c(x)u(y), u(x)c(y), s(x)c(y), c(x)s(y), c(x)c(y)\}$$

where x and y vary over the domain $\{-1, 0, +1\}$ of our 3×3 neighborhood, and where

$$u(x) = \sqrt{\tfrac{1}{3}}, \quad s(x) = \sqrt{\tfrac{1}{3}} \sin\left(\tfrac{1}{3}\pi x\right), \quad c(x) = \sqrt{\tfrac{2}{3}} \cos\left(\tfrac{2}{3}\pi x\right)$$

The polynomial basis $\{1, x, y, xy, 2-3x^2, 2-3y^2, (2-3y^2)x, (2-3x^2)y, (2-3x^2)(2-3y^2)\}$ can also be used to express this un-normalized discrete cosine basis. We unfold the 3×3 matrix into a single 9-dimensional vector $b = [f_{i-1,j-1} \quad f_{i,j-1} \quad f_{i+1,j-1} \quad \cdots \quad f_{i+1,j+1}]^T$ of sample values, and write the cosine basis as a 9×9 orthogonal matrix B, where the

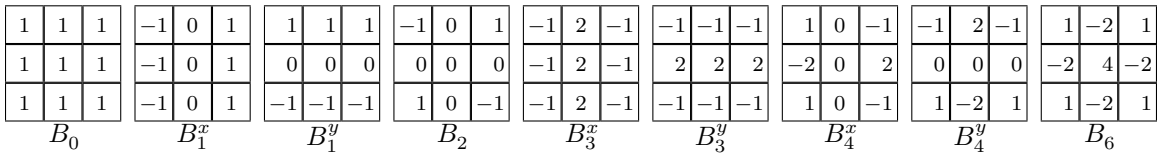


Figure 16: Basis functions for the 2D discrete cosine transform (not normalized).

columns of B are the basis functions. Then the forward discrete cosine transform is simply $x = B^T b$, with x being the DCT coefficients in order of increasing frequency.

To extend the ideas of Isenburg et al. from 1D to 2D, we must rank the basis functions by increasing frequency. The cosine basis formulation gives us pairs of frequencies (ν_x, ν_y) for the horizontal and vertical direction, which must be consolidated into single frequencies. We approach this by deriving the cosine basis through eigenanalysis of the symmetric combinatorial graph Laplacian $\mathcal{L}$ (also called the Kirchoff matrix [109])

$$l_{ij} = \begin{cases} \deg(i) & \text{if } i = j \\ -1 & \text{if } i \text{ and } j \text{ are adjacent} \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

where we consider the graph formed by the 3×3 neighborhood in isolation, with vertical and horizontal edges between adjacent samples. Here $\deg(i)$ denotes the degree or number of neighbors of a sample i, e.g. $\deg(i)$ is two for corner samples, three for edge samples, and four for face samples. As noted by Taubin [97], the eigenbasis of the normalized (asymmetric) Laplacian coincides with the Fourier basis, and the eigenvalues $\{\lambda_i\}$ of $\mathcal{L}$ correspond to frequencies. The above un-normalized (symmetric, positive semidefinite) Laplacian $\mathcal{L}$ has real non-negative eigenvalues $\{0, 1, 1, 2, 3, 3, 4, 4, 6\}$ and the cosine basis as eigenbasis. We will use B_λ to denote the eigenvector (i.e. basis function) with corresponding eigenvalue λ, and B_λ^x and B_λ^y to distinguish pairs of eigenvectors with equal eigenvalues (Figure 16).

Our formulation shows that there is a unique highest frequency $\lambda = 6$ with associated basis function B_6. Given the eight known samples in the bi-Lorenzian and radial predictors, similarly to Isenburg et al. we set the highest frequency x_6 to zero and solve for the unknown sample as a linear combination of the $m = 8$ known samples, which results in the weights given in Equations 11 and 12 for corner and center predictions. When $m < 8$, a similar

strategy is possible by zeroing $9 - m$ of the highest frequencies. However, we may need to resolve two issues:

1. The $9 - m$ first basis functions may not form a basis for the set of known samples, e.g. $\{B_0, B_1^x, B_1^y\}$ is not a basis for $b = [f_{i-1,j-1} \quad f_{i,j-1} \quad f_{i+1,j-1} \quad 0 \quad \cdots \quad 0]^T$.

2. In situations when only one of B_λ^x and B_λ^y is needed (e.g. when exactly two samples are known), we may reduce the total frequency response by choosing a linear combination of B_λ^x and B_λ^y.

Let M be an $m \times n$ mask matrix that extracts the m known samples Mb from b, i.e. each row of M has a single one entry and remaining zeros. We wish to solve the underconstrained system $MBx = Mb$ for x with as many high frequencies of x zeroed as possible. This can be done via linearly constrained least-squares methods, which involves symbolic inversion of an $(m + n) \times (m + n)$ matrix [61]. We here show how to accomplish the same goal via inversion of a smaller $m \times m$ matrix.

To achieve our goal, we need to find an m-dimensional basis for Mb by selecting from or combining the $n > m$ column vectors MB. When excluding a vector from MB, we implicitly zero the corresponding frequency response. Our approach is to incrementally construct an $n \times m$ interpolation matrix P that linearly combines vectors from MB such that $MBPy = Mb$ is a fully constrained system of m equations, with $x = Py$. We achieve this by adding to P columns that select basis functions from MB in order of increasing frequency. If a basis function projected onto the space of known samples is redundant (linearly dependent) with respect to the partially constructed basis, we exclude it and consider the next basis function. When we encounter an eigenspace, i.e. two basis functions with the same eigenvalue, one of three situations arises: (1) The whole eigenspace is redundant, and we exlude it. (2) The whole eigenspace is nonredundant, and we include it. (3) The eigenspace is partially redundant, in which case we first "rotate" the eigenspace by an angle θ until one of the rotated basis functions becomes redundant. (Note that any rotation of an eigenspace preserves eigenvalues and orthogonality with the rest of the basis.) This leaves a nonredundant basis function $B_\lambda^\theta = \cos(\theta)B_\lambda^x + \sin(\theta)B_\lambda^y$ and we add to P a column that

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \qquad P^T = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{2}{\sqrt{5}} & \frac{1}{\sqrt{5}} & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

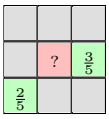


Figure 17: Example mask matrix M, interpolation matrix P, and predictor weights.

has $\cos(\theta)$ and $\sin(\theta)$ in the rows corresponding to B_λ^x and B_λ^y. The effect of this rotation is to "align" the basis function with the spatial configuration of known samples. One can show that this rotation leads to the minimal total frequency response $||x||$.

We can now compute the solution $x = P(MBP)^{-1}Mb$ using matrix inversion. We are, however, not interested in the frequency response x but in the weights of the known samples Mb, and hence we apply the inverse DCT to x and compute $Bx = Wb$ where W is the $n \times n$ weight matrix $W = BP(MBP)^{-1}M$. The weights W are then used to predict unknown samples in b from known ones.

We have implemented this method symbolically in Mathematica and computed exact weights for all neighborhood configurations, resulting in 41 unique weights in the range $[-4, +4]$ that are predominantly integers and otherwise rationals. For the concrete case of a neighborhood of 3×3 in 2D, the table takes a total 2304 positions, each position has 8 weights, totaling 18 kbytes of memory. This table has redundancy that could be eliminated through symmetry and rotation, but due to its already small size, we do not recommend doing it for the 3×3 2D case. For ease of use we recommend to have the table of spectral predictors packaged with the compressor and the decompressor. This list of weights can be found at the appendix. Note that our weights always add to unity, making our predictor affine invariant.

4.4.5.2 Choosing a neighborhood

Via translation we can form nine 3×3 neighborhoods around each predicted sample p, see Figure 18. Depending on the configuration of known samples it is not immediately clear which neighborhood to predict from. For example, in Figure 10, we can appreciate several stencils to predict the point between numbers 42 and 27. We propose training the compressor on the given data set: each of the 9×2^8 predictors is exercised on each sample and receives a ranking based on its mean error. This short ranking is transmitted before

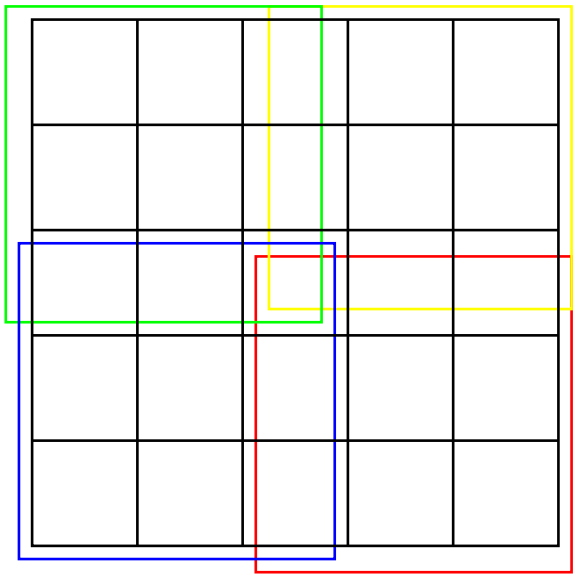


Figure 18: The set of 3×3 neighborhoods to predict a point results in an area of 5×5 from which to choose the adequate stencil.

compression begins and determines the choice of the predictor.

In case that producing and transmitting the training is not an option, we recommend to use the Spectral predictor with more known points. We have compared Spectral predictors of different known points on several datasets (see Figure 19). Consistently the spectral predictor with more points performs better; whether the Spectral predictor had more close or more far points did not affect its performance. We have concluded that limiting ourselves to immediate neighbors of close points is not an effective method to choose the best Spectral predictor for a situation. A Spectral predictor with more known points approximates a higher degree function as compared to a Spectral predictor with less known points that approximates a lower degree function; this results in an increase in predictor accuracy for well behaved datasets.

In Figure 20 we use random sampling of two data sets to show that our approach improves upon several alternatives that we have explored:

- **Spectral Best**: From all available spectral predictors, the Spectral Best method uses the most accurate one. It is not possible for the decoder to know which is the best predictor, but it provides a bound of the effectiveness of the spectral predictors.

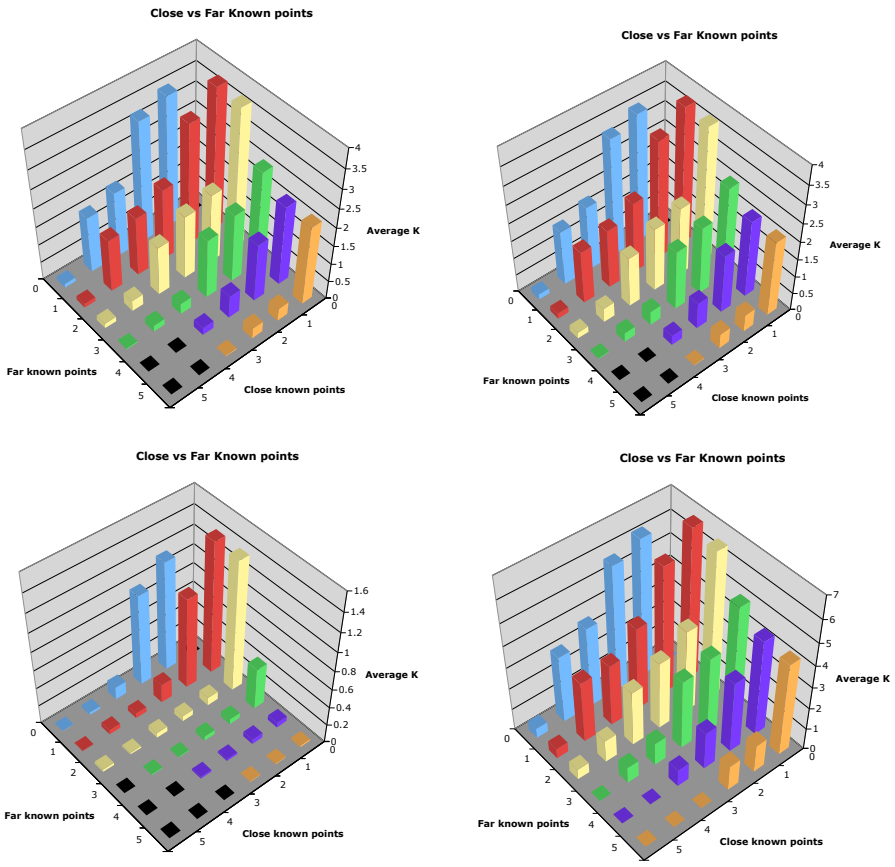


Figure 19: Comparison of Spectral predictors depending on their number of close and far known neighbors. From top to bottom, left to right, the datasets used are: Velocity-X, Velocity-Y, Pressure and Synthetic.

- **Spectral Trained**: For each dataset, the spectral predictors have been ranked based on their precision. This method adapts the spectral predictors to the idiosyncrasies of each dataset. It has a small overhead of the preference between spectral predictors.

- **Spectral Average:** This method computes the spectral predictor for each different neighborhood and selects the average of all the predictions.

- **Spectral Median**: Similar to spectral average, this method returns the median of all spectral predictors.

- **Spectral Local**: This is the spectral predictor applied only to the closest neighbors, to exploit locality.

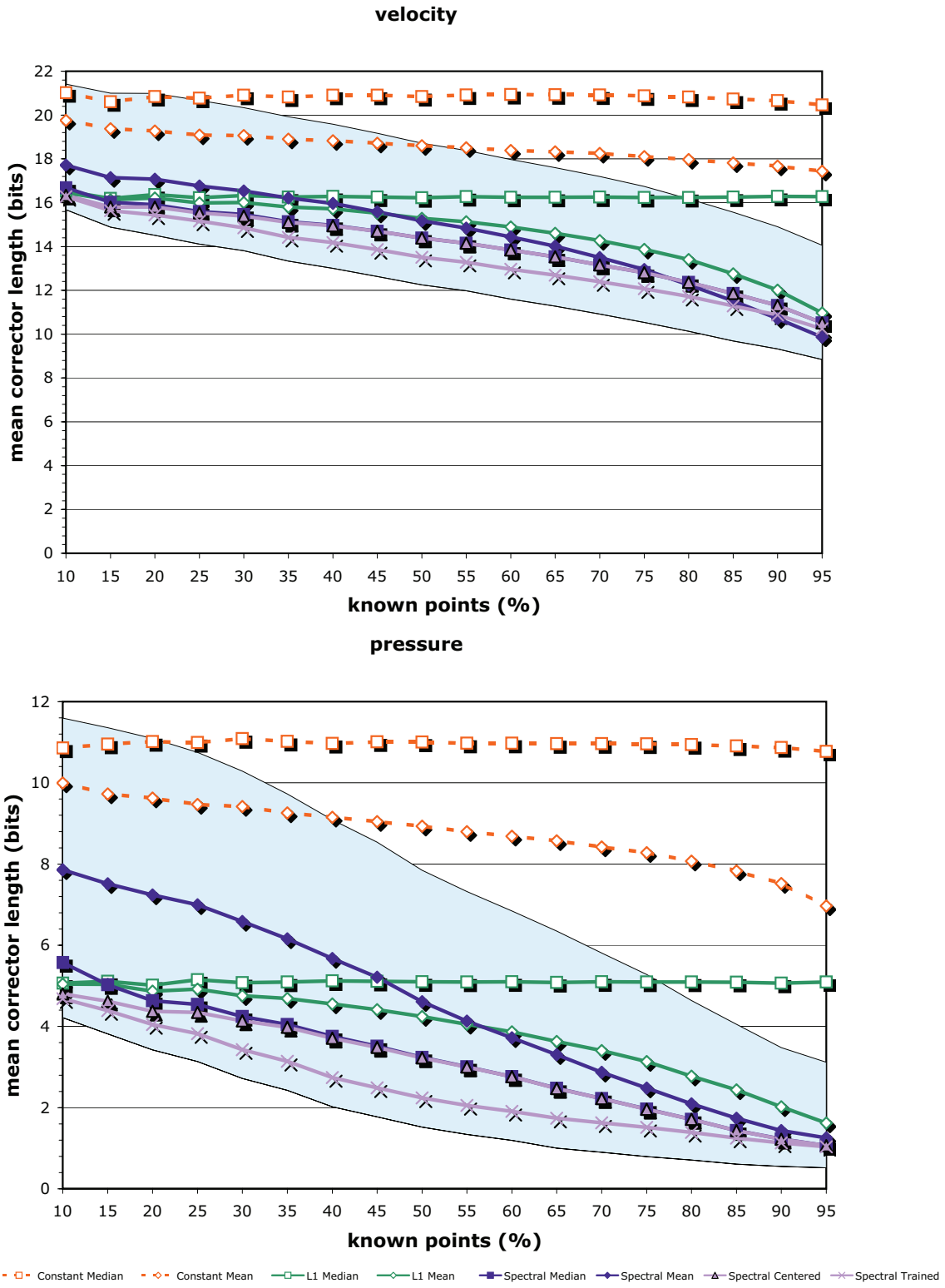


Figure 20: Predictor quality as a function of number of known points in the 3×3 neighborhood. The shaded area indicates the range between best and worst spectral prediction.

- **Spectral Worst**: This method always returns the spectral predictor with a larger error. It provides a worst bound for the effectiveness of the predictor.
- **L1**: This is a basic parallelogram predictor, applied in 4 contiguous neighborhoods. It returns the median of all the parallelograms.
- **Linear Median**: This predictor returns the median of all possible groups of 3 points that fit a plane across the predicted point.
- **Constant Median**: This predictor returns the median of the known values around our predicted point.

Figure 20 shows that there is an improvement through training of the spectral predictors. Training only requires pre-computation and storing a list ranking the spectral predictors; it is an improvement without any extra cost for subsequent compressions.

4.5 Extending the prediction to higher dimensions

The predictors discussed so far have been applied to regular grids in 2D. For regular grids in 3D, we can apply our 2D predictors slice per slice of the dataset. This approach has the benefit of using existing methods and the benefit of requiring to hold in memory no more than a slice of the data (even less with small footprints). The main drawback of using 2D predictors in a 3D dataset is that coherence across the third dimension is not exploited, thus achieving suboptimal compression.

The Lorenzo predictor scales to any dimension. The difference from 2D to 3D in the Lorenzo predictor is the difference between fitting a plane (2D) or a quadratic function (3D). Several tests have shown improvements of 7 to 29 % when using a predictor more fitted for the data (3D instead of 2D). For the Diffusivity dataset, shown in Figure 28, 2D Lorenzo predictor in the XY plane reports 20.6 bits per symbol versus the 19.2 bits per symbol of the 3D Lorenzo predictor. On the Pressure data, 2D Lorenzo predictor applied on the XY plane reports 6.88 bits per symbol versus 4.79 bits per symbol of the 3D Lorenzo predictor, a 29% improvement.

We must note that it is not always the case that a predictor in higher dimensions outperforms a predictor in a lower dimension. Predictor performance is very dependent on the characteristics of the data; it is possible to have data, such as a dataset with a low signal to noise ratio that achieves better compression with a 2D predictor than a 3D one because the prediction in 3D is more susceptible to noise. Even so, with few exceptions, the use of a predictor with the same dimensionality as the data is an improvement.

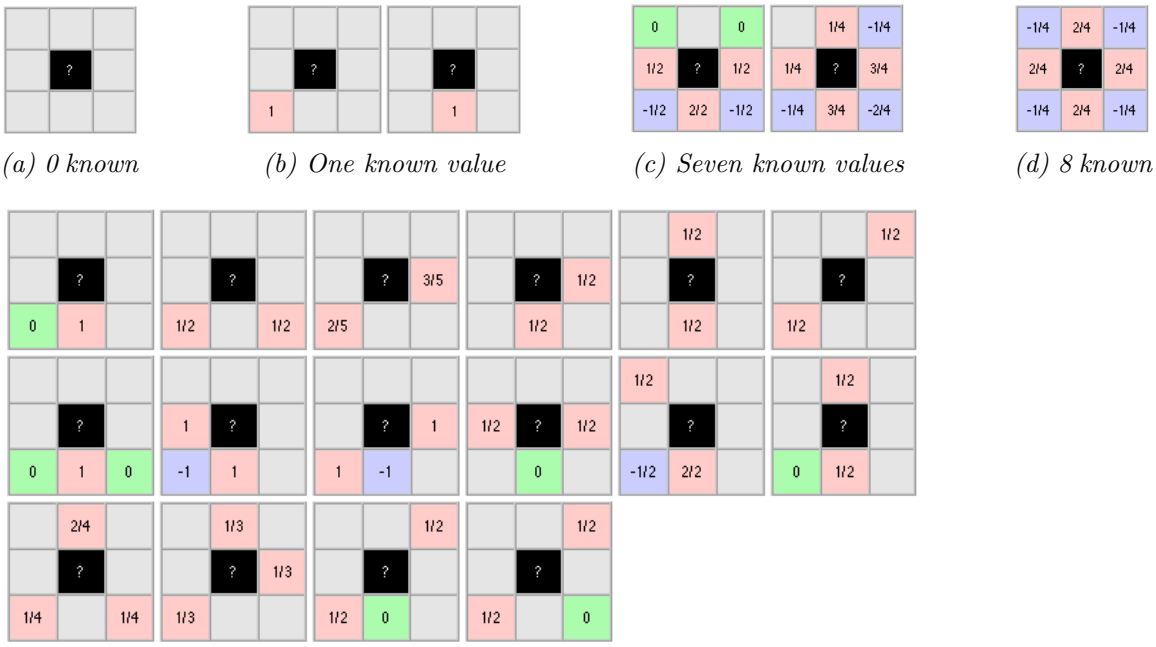
4.5.1 Spectral Extension to 3D

The mathematical derivation of the Spectral predictor can be applied to any neighborhood in any dimension; in this sense Spectral has a very straightforward theoretic extension to 3D. In practice however, the challenge to extend Spectral prediction to 3D stems from the number of stencils required. Computing the Spectral weights is computationally intensive, but only needs to be done once. For a stencil with n points, the table will have $n(n-1)2^{n-1}$ weights stored on it. In the 2D case of a 3×3, n is 9, and there are 18432 weights to store. In 3D, a $3 \times 3 \times 3$ stencil has $n = 27$ samples, the size of the table is $47,110,422,528$, and even if we only needed one byte to store each weight, the table would need close to 44 Gbytes.

Table 1: For each position of the predicted point in Spectral 3D, the number of distinct configurations is shown. Volume refers to the case where the predicted point is in the center of the 3D cube, Face refers to the case where the predicted point is in the center of a face of the 3D cube, Edge refers to the case where the predicted point is in the center of an edge of the 3D cube, and Vertex refers to the case where the predicted point is on a vertex of the 3D cube.

	number of different predictors
Volume	1,426,144
Face	8,456,256
Edge	16,846,848
Vertex	11,250,688

Such a table is too large to be included in both the encoder and the decoder. We have explored ways to reduce the size of the table. Many predictors stored in the table are replicated, due to symmetry and rotation. As an example, Figure 21 shows the spectral



(a) 0 known *(b) One known value* *(c) Seven known values* *(d) 8 known*

(e) Two and Three known values

(f) Four known values

(g) Five and Six known values

Figure 21: Weights for all the distinct spectral predictors in 2D where the predicted point is in the center. The figures are divided by the number of known values, from zero to eight. From a total of 256 possibilities the number can be reduced to 51.

predictors in 2D where the predicted point is in the middle. By eliminating redundant entries from 256 different predictors we can reduce the table to 51.

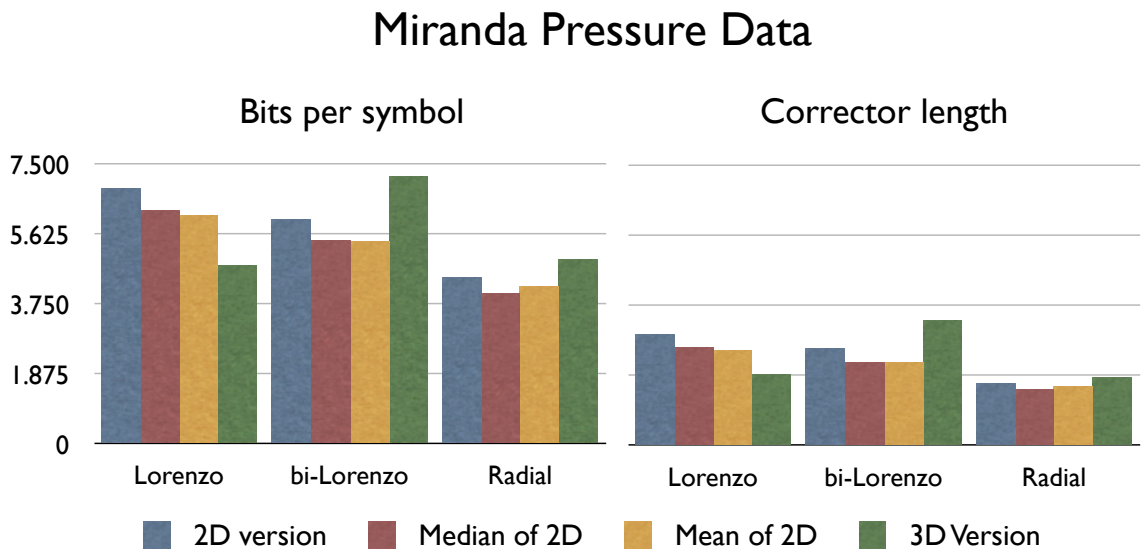


Figure 22: Comparison of 2D predictors with their average over the three axes and with their 3D counterparts. The left graph shows the improvement in compression, the right graph shows the improvement in predictor accuracy. The test was performed in the Pressure data from the Miranda dataset, from the Lawrence Livermore National Laboratory.

We have applied redundancy elimination to the Spectral table in 3D, and reduced the number of weights needed from $47,110,422,528$ to $18,539,482$. The decomposition of weights by type of predictor is shown in Table 1. Even with this reduction, and using a byte for each weight, the reduced spectral table needs 18.4 Mbytes. To access the reduced table it is necessary to map the current stencil to a canonical one, which increases the computational cost of retrieving the spectral weights with respect to the straightforward method in 2D.

For most applications, an encoder/decoder that by itself takes around 20 Mbytes, and is prohibitive because page faults when accessing each predictor would decrease the speed of compression and decompression. As discussed earlier, one can explore a variety of approaches to reduce the size of the spectral table, such as, through the use of training pick the best and most used predictors, then encode the data using only the selected predictors; if a needed predictor was not selected, fall back to simpler spectrals. This approach banks on the fact that only a subset out of the large number of spectral predictors is actually

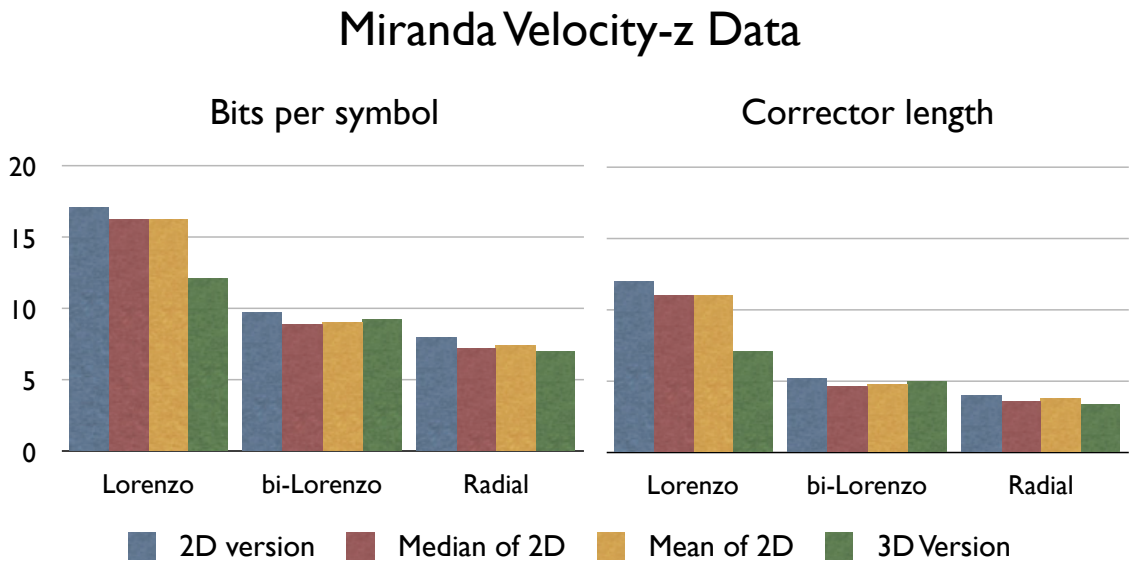


Figure 23: Comparison of 2D predictors with their average over the three axes and with their 3D counterparts. The left graph shows the improvement in compression, the right graph shows the improvement in predictor accuracy. The test was performed in the Velocity-z data from the Miranda dataset, from the Lawrence Livermore National Laboratory.

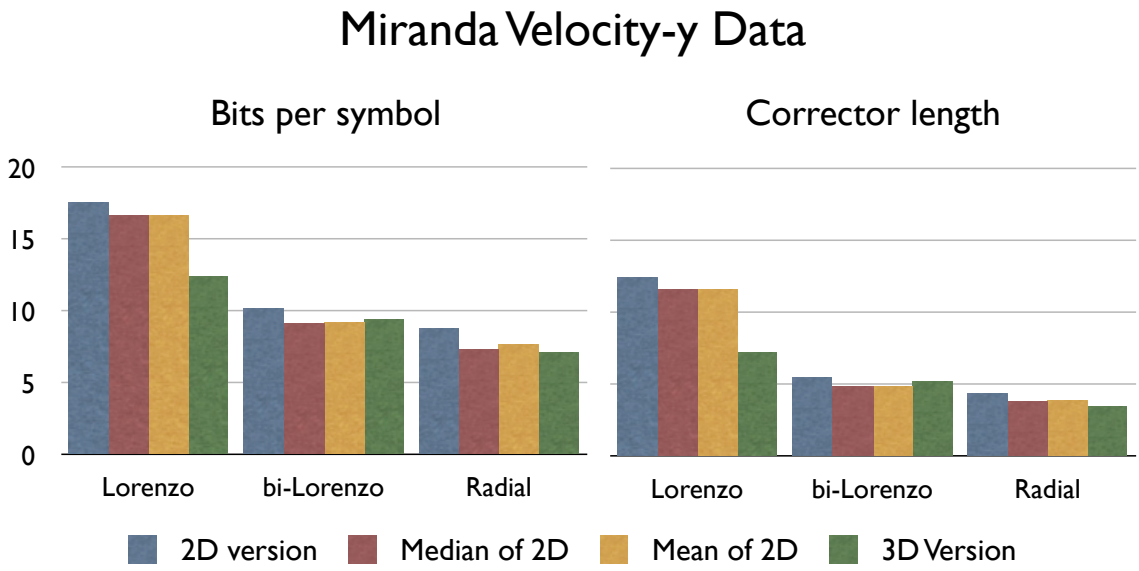


Figure 24: Comparison of 2D predictors with their average over the three axes and with their 3D counterparts. The left graph shows the improvement in compression, the right graph shows the improvement in predictor accuracy. The test was performed in the Velocity-y data from the Miranda dataset, from the Lawrence Livermore National Laboratory.

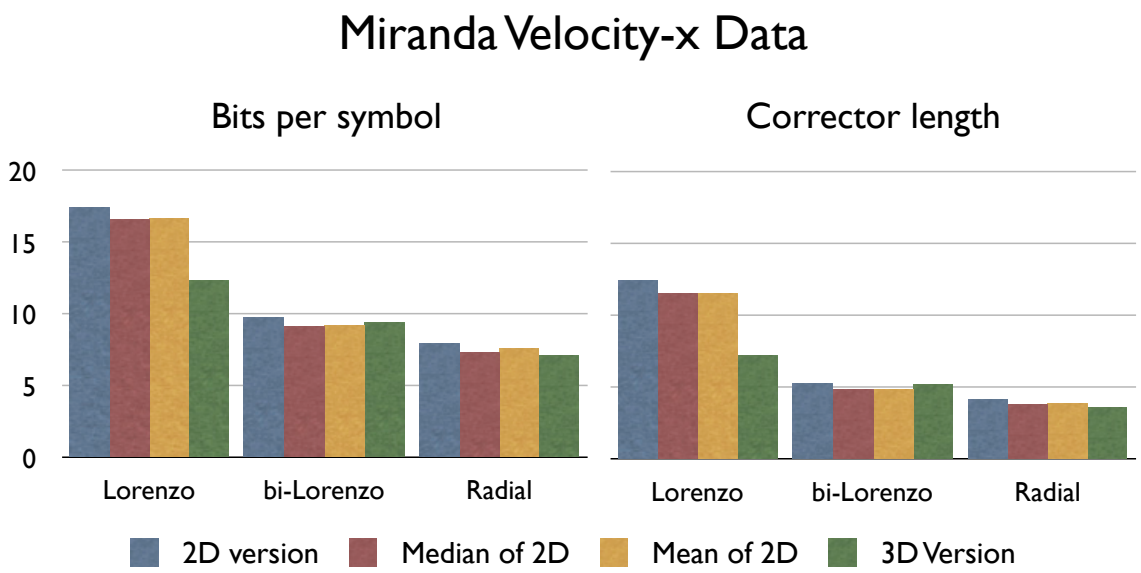


Figure 25: Comparison of 2D predictors with their average over the three axes and with their 3D counterparts. The left graph shows the improvement in compression and the right graph shows the improvement in predictor accuracy. The test was performed in the Velocity-x data from the Miranda dataset, from the Lawrence Livermore National Laboratory.

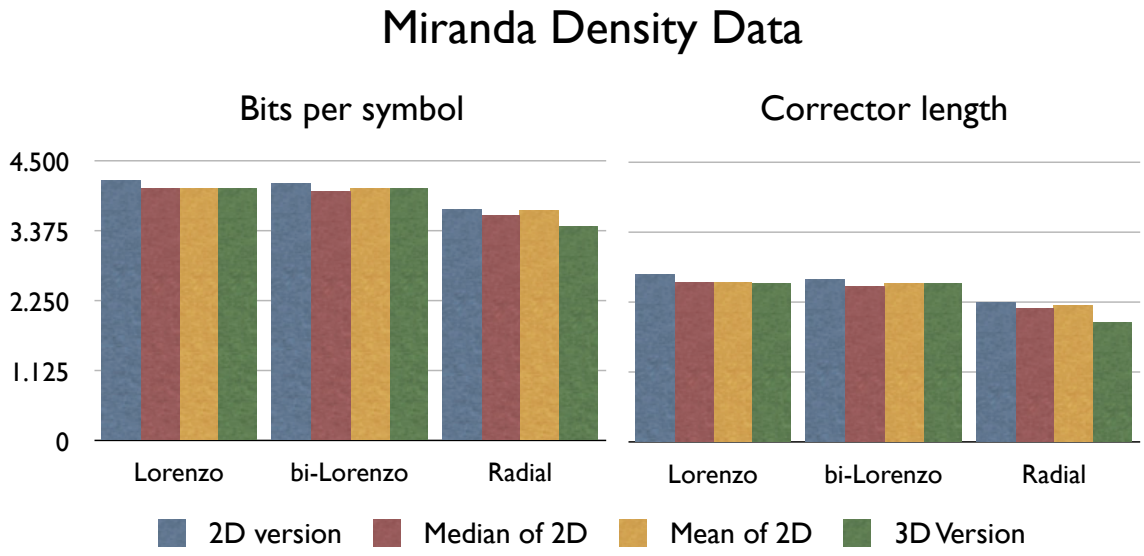


Figure 26: Comparison of 2D predictors with their average over the three axes and with their 3D counterparts. The left graph shows the improvement in compression, the right graph shows the improvement in predictor accuracy. The test was performed in the Density data from the Miranda dataset, from the Lawrence Livermore National Laboratory.

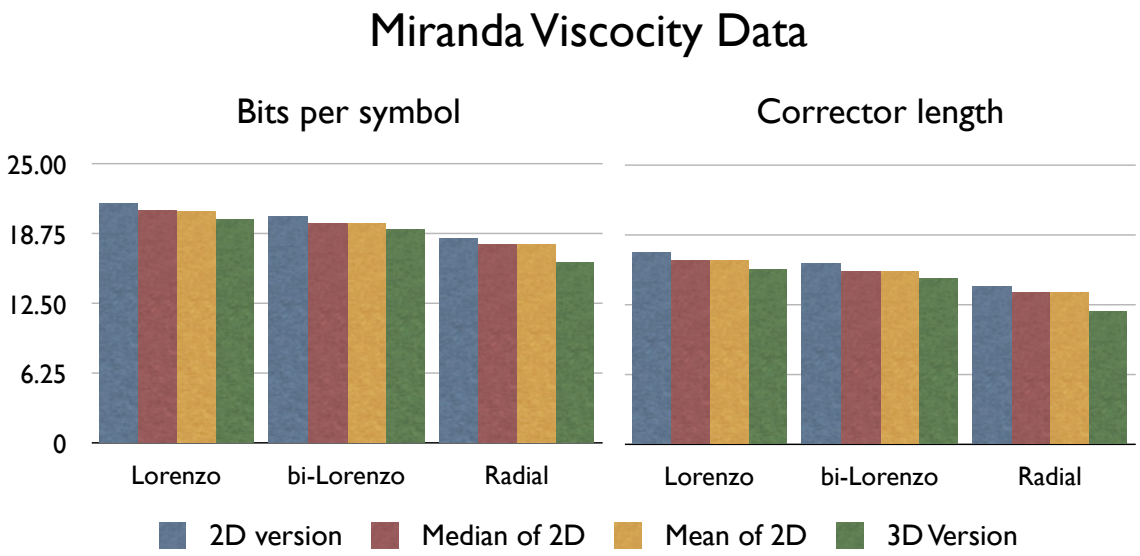


Figure 27: Comparison of 2D predictors with their average over the three axes and with their 3D counterparts. The left graph shows the improvement in compression, the right graph shows the improvement in predictor accuracy. The test was performed in the Viscocity data from the Miranda dataset, from the Lawrence Livermore National Laboratory.

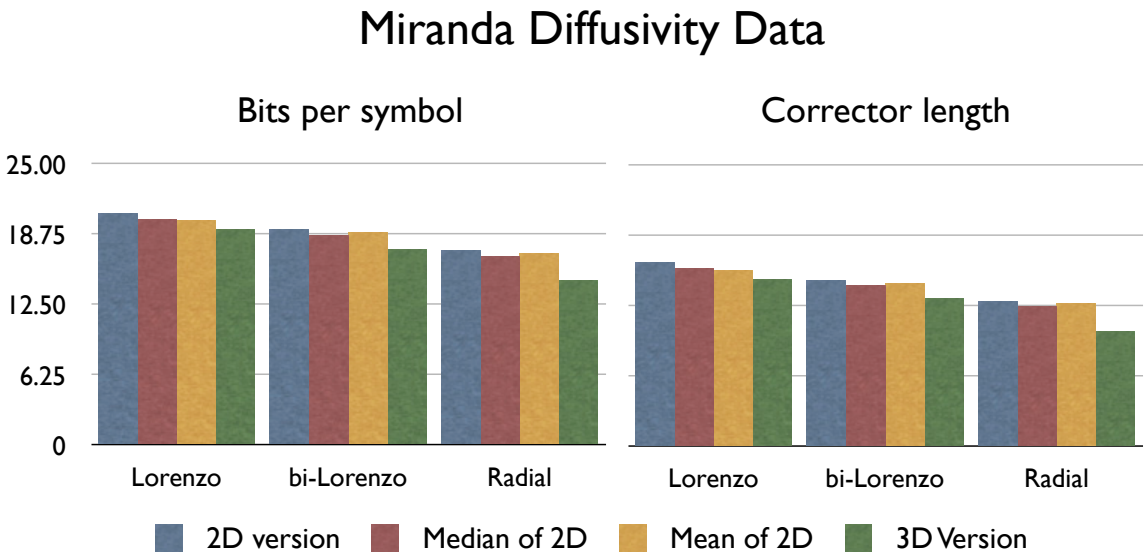


Figure 28: Comparison of 2D predictors with their average over the three axes and with their 3D counterparts. The left graph shows the improvement in compression, the right graph shows the improvement in predictor accuracy. The test was performed in the Diffusivity data from the Miranda dataset, from the Lawrence Livermore National Laboratory.

used. The subset of used predictors depends on the data and the used traversal; this is a venue for future work.

We perform a series of tests comparing several 2D predictors to their 3D counterparts. We compare the Lorenzo Predictor, the bi-Lorenzian and the Radial in 2D against the mean of three 2D axes aligned predictors in a 3D dataset, and against a pure 3D version of the predictors. We have tested the predictors in several 3D scientific datasets (from the Miranda data [23]). The datasets from the Miranda experiment have $288 \times 288 \times 1152$ size.

We compare the 2D version (applied on the XY plane) of the Lorenzo, bi-Lorenzian and Radial predictors with the average of applying the 2D prediction on the XY, XZ and YZ axis, with the mean of the former predictors, and with the 3D versions of such predictors. It is important to note that the mean of the Spectral prediction in the XY, XZ and YZ plane is the Spectral predictor for 3D for that stencil. Our results, in Figures 22 to 28 show that there is a significant improvement by extending a predictor to 3D, and it becomes more significant with the number of points used. Lorenzo Prediction for the Diffusivity dataset needs 20.5 bits per symbol using the 2D version of the predictor, versus 19.2 bits per symbol in the 3D case, a 6% improvement. For the same dataset, Radial 3D reports 14.6 bits per symbol versus 17.2 bits per symbol reported by the 2D version of the Radial predictor, an improvement of 15%. Improvements of more than one bit per symbol are significant. There is almost no difference between the median and mean of the three 2D spectral predictors. The mean is a spectral predictor by itself, while the median (also a spectral predictor, but only in 2D) is more resistant to noise due to its smaller stencil.

Going to 3D is an important improvement. Yet, given the challenge of full extension of Spectral predictors to 3D, we have chosen to apply Spectral 2D for our implementation of irregular stencil prediction. We consider Spectral prediction in 2D on each of the 3 possible axes, and pick the stencil with more points.

4.6 Applications and Results

We have applied successfully our techniques to the compression of scientific simulations, on a dataset simulating a house on fire for firemen in training. Our predictors can be applied

to any regular grid data. Medical data requiring lossless compression can benefit from our techniques, as well as 2D data such as digital elevation maps and regular images.

We evaluate predictor performance in terms of the number of significant corrector bits, which is the dominating cost in predictive coders for high-precision data [28, 68, 85]. For floating point data, we compute an integer corrector that measures the number of distinct representable floating point values separating the actual and predicted value (see [68]).

In the following subsections we describe our implementations and results.

4.6.1 Scanline compression algorithm, using Lorenzo Prediction

Consider a 4D scalar data set organized in an array $F[xmax, ymax, zmax, tmax]$. The pseudocode for the Lorenzo compression algorithm is presented in Figure 29. The variable d indicates the dimension of the predictor; $x_1, \ldots, x_n$ are the coordinates of a sample in the data, $dmax$ is the number of samples in the d^{th} dimension, and k denotes the greatest index between 1 and n where x_k equals -1.

$Lorenzo(d, x_1, \ldots, x_n)$
  if all $x_1, \ldots, x_n$ differ from -1
    $E := LorenzoPredictor(d, x_1, \ldots, x_n)$
    $Encode(F[x_1, \ldots, x_n] - E)$
  else
    $Lorenzo(d-1, x_1, \ldots, x_{k-1}, 0, x_{k+1}, \ldots, x_n)$
    for $i = 1$ to $dmax$ do
      $Lorenzo(d, x_1, \ldots, x_{k-1}, i, x_{k+1}, \ldots, x_n)$

Figure 29: Pseudocode for Lorenzo predictor based compression on a 4D dataset.

The function *LorenzoPredictor* computes the Lorenzo predictor of dimension d (the first parameter) at the coordinates $x_1, \ldots, x_n$. The starting call to compress the 4D dataset is:

$$Lorenzo(4, -1, \ldots, -1)$$

For example, consider the 2D case of a 3×3 matrix H with points from $(0, 0)$ to $(2, 2)$. The trace of the compression program on the integer lattice is described in Figure 30.

call	*encoded value*
$Lorenzo(2,-1,-1)$	
$Lorenzo(1,-1,0)$	
$Lorenzo(0,0,0)$	$H[0,0]$
$Lorenzo(1,1,0)$	$H[1,0]-H[0,0]$
$Lorenzo(1,2,0)$	$H[2,0]-H[1,0]$
$Lorenzo(2,-1,1)$	
$Lorenzo(1,0,1)$	$H[0,1]-H[0,0]$
$Lorenzo(2,1,1)$	$H[1,1]-(H[1,0]+H[0,1]-H[0,0])$
$Lorenzo(2,2,1)$	$H[2,1]-(H[1,1]+H[2,0]-H[1,0])$
$Lorenzo(2,-1,2)$	
$Lorenzo(1,0,2)$	$H[0,2]-H[0,1]$
$Lorenzo(2,1,2)$	$H[1,2]-(H[0,2]+H[1,1]-H[0,1])$
$Lorenzo(2,2,2)$	$H[2,2]-(H[2,1]+H[1,2]-H[1,1])$

Figure 30: Trace of the encoding of the Lorenzo Predictor based scanline compression on a small 2D dataset.

The footprint used by the Lorenzo predictor for compressing or decompressing a dataset is the size of a single $(n-1)$-dimensional slice, as illustrated in Fig. 31 for the $\mathbb{R}^2$ case and in Fig. 32 for the $\mathbb{R}^3$ case.

The footprint for compressing and decompressing a dataset of size D^n is $D^{n-1}+D^{n-2}+\cdots+D+1$. If all the dimensions do not have the same length, the size of the footprint depends on the traversal order, which can be chosen to minimize the size. The footprint is implemented as a circular FIFO queue.

We compress the corrections with an adaptive arithmetic encoder, this makes us store a probability table in memory. While this space is generally much smaller than the whole data set, it is often not necessary to store the probability for every possible correction. If memory is scarce, only the frequently occurring, small corrections around zero (Fig. 34) need to be compressed, whereas occasional large corrections can be flagged and transmitted verbatim, with minimal impact on the compression rate.

When lossy compression is acceptable, we allow a small discrepancy between the compressed data and the real data in order to improve the compression ratio. We have considered two different error metrics: L_∞ (maximum error) and L_2 (root mean square error).

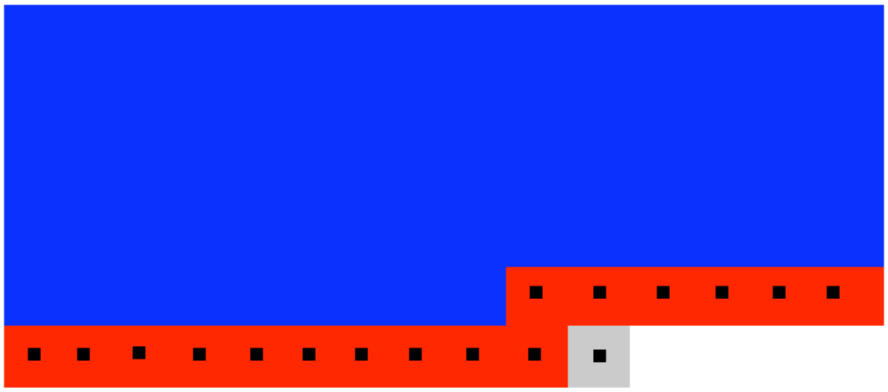


Figure 31: Lorenzo Prediction's footprint in 2D. When compressing an $\mathbb{R}^2$ dataset, the next value (grey square) is predicted by using values from the footprint (red). The other previously processed values (blue) are not used by the predictor and need not be kept in memory.

L_∞ is the maximum of the errors in the decompressed version, L_2 is the square root of the sum of the squares of the errors. L_∞ is a metric used when it is imperative that the error never exceeds a threshold. L_2 metric allows a small number of errors to be large while the average error remains small.

To guarantee that we do not exceed a prescribed L_∞ error, we quantize the residuals from our predictor and readjust the values at the scalar field to compensate for any possible error accumulation. Thus the error made is at most the quantization error. The adjusted corrections are encoded in a lossless fashion.

We have tested the Lorenzo predictor based compression approach both on synthetic and real data. Using synthetic data, we populated a volume dataset with functions of a degree higher than that of the predictor. When Lorenzo predictor in 3D is applied to a volume dataset filled with a cubic function, the predictor makes a relative error between 3 and 8% (measure taken from applying the predictor to different cubic functions), while a 4D predictor against a volume set filled with a quartic function makes a relative error of less than 1%. Both errors are measured in the L_∞ sense.

We have also tested our predictor on two real 4D datasets. We use two different lossless encoding methods to write the corrections to disk. The first one is to feed the corrections

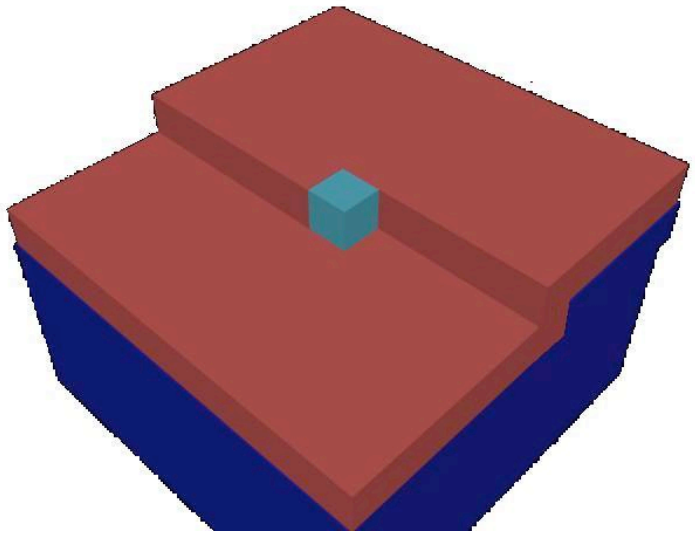


Figure 32: Lorenzo Prediction's footprint in 2D. When compressing an $\mathbb{R}^3$ data set, the next value (light cube in the center) is predicted by using values from the footprint slice (red). The other previously processed values (bottom in blue) are not used by the predictor and need not be kept in memory.

to an adaptive arithmetic encoder. The second one uses a context arithmetic encoder (like the one studied by Bell [13]) with the actual prediction as the context for the correction. The context arithmetic encoder yields a 25% gain over the adaptive arithmetic encoder. We study the benefits of using 4D compression rather than a series of 3D compressed slices. All values are quantized to one byte.

Our first data set, courtesy of Professor Chris Shaw from Georgia Tech, is a 3D model of a house on fire, which shows how the fire, smoke and pressure progress through time and space. This data set has a large number of zones where the scalar values are uniform, and zones that exhibit high gradients and correspond to the moving front of the fire. Because of the large number of 0 corrections in the data set, best results are obtained using a lossless RLE compression method, which we applied both to the 3D and 4D residuals for comparison. The uncompressed 4D data set has a size of $43,202,395$ bytes and a total information content (based on its entropy) of $23,491,302$ bytes. Compressing each 3D slice independently we obtain $1,076,587$ bytes (2.49% of the total), and $524,768$ bytes (1.21% of the total) using 4D compression. Due to the high coherence in all dimensions, the 4D predictor outperforms the 3D compression applied to individual slices by 50%.

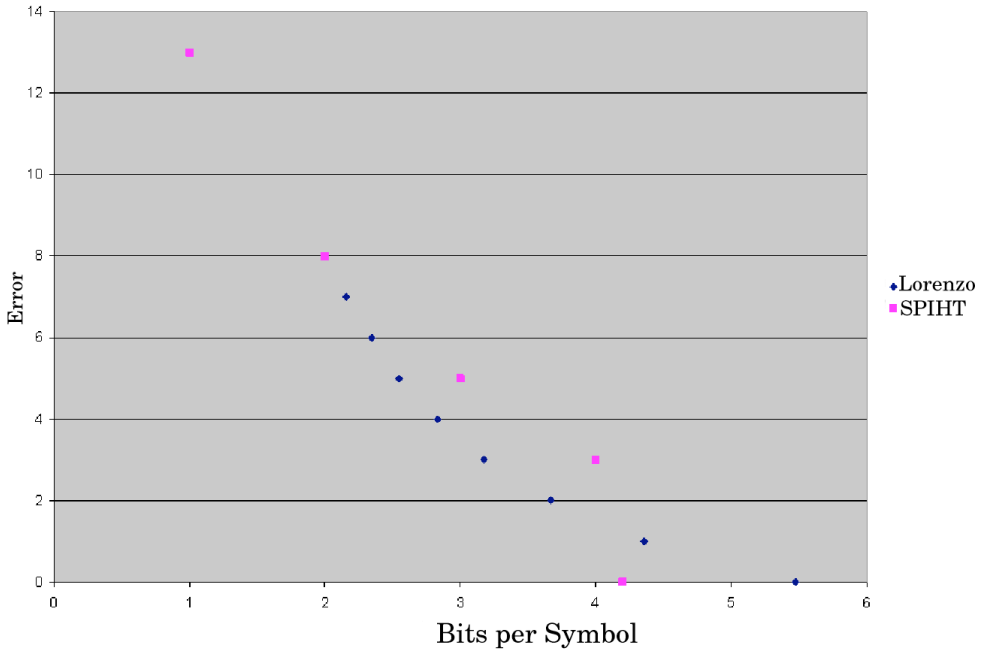


Figure 33: L_∞ comparison of the Lorenzo Predictor (blue) and SPIHT (pink), a 2D wavelet image compressor.

Our second test data set was produced in a fluid mixing simulation at Lawrence Livermore National Laboratory, as described in [77]. A volume rendered image of this data set is shown in Figure 9. During the beginning time steps the fluids are virtually at rest and the data set is relatively easy to compress. As the fluids mix up in later time steps, the compression ratio decreases. For this data set, we chose to use an adaptive context arithmetic encoder as a postprocessing step to the Lorenzo prediction.

The 3D time slice predictor produces the best compression on this data set. 3D predictors for 3D slices in all the other directions are less effective. To explain this behavior, consider that this data set was originally computed at high resolution in time (27,000 time steps were simulated), but then decimated and quantized due to limited storage. The floating point data was quantized to one byte and only one frame out of every hundred was actually stored, although no decimation was applied in the spatial dimensions. This subsampling phase has significantly reduced the coherence along the time axis. Our approach yields $304,937,058$ bytes using the 3D Lorenzo predictor on each time slice (1.77 bits per symbol) and $318,871,620$ bytes using 4D prediction (1.85 bits per symbol).

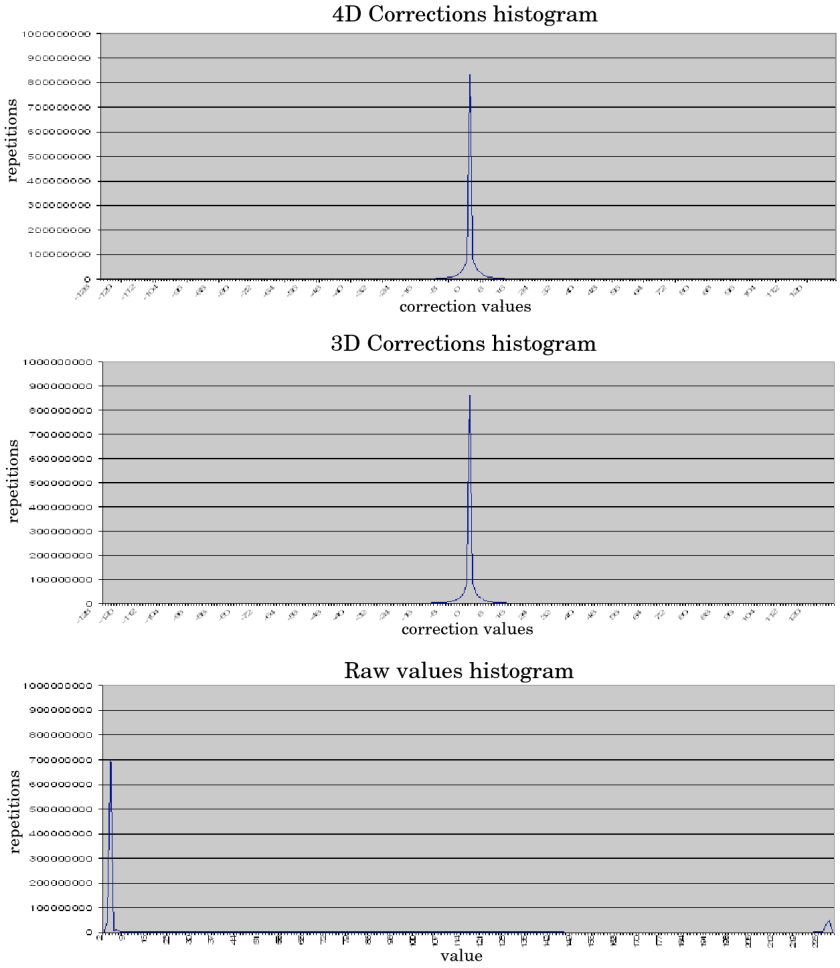


Figure 34: Histograms for the LLNL data set, compressed with Lorenzo Predictor. Top: Frequency of the 4D corrections (which range from 0 to 1,000,000,000) as a function of their value, ranging from -128 to 127. Middle: Frequency of the 3D corrections for a single time slice. Bottom: Raw values, ranging from 0 to 255.

We have compared the Lorenzo predictor with wavelets in two scenarios. In the first case we evaluate lossless compression, where we compare cubic wavelets with the Lorenzo predictor for several 4D data sets. As can be seen in Table 2, which reports the entropy of the corrections of both schemes, wavelets and the Lorenzo predictor produce comparable results. In our second scenario, the Lorenzo predictor was compared to SPIHT [92], an efficient wavelet coder that uses the S+P transform, in terms of rate distortion using lossy compression. Figure 33 shows that the Lorenzo predictor performs better in the L_∞ sense on this data set. The compression ratios of this example were computed by compressing the residuals using a context arithmetic encoder for the Lorenzo predictor, whereas SPIHT

Table 2: Entropy of the residuals produced by wavelets and the Lorenzo predictor for a 4D data set. No quantization or truncation of the data or residuals was done.

Dataset	4D Lorenzo Predictor	Cubic Wavelets
Smooth 64^4	0.16 Bits/Symbol	0.20 Bits/Symbol
Rough 64^4	3.73 Bits/Symbol	3.28 Bits/Symbol
Rough 128^4	1.75 Bits/Symbol	1.80 Bits/Symbol

used its hierarchical tree method.

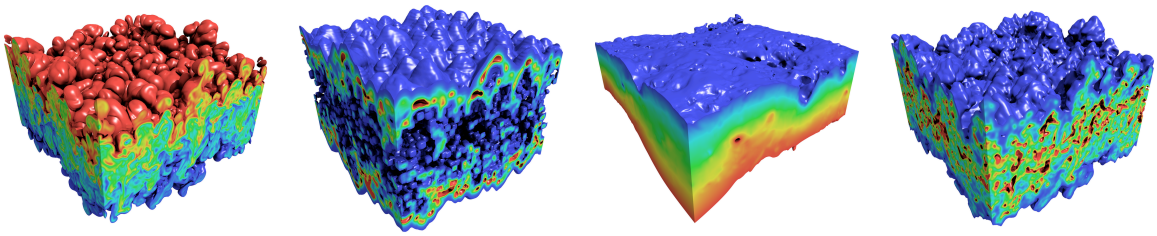


Figure 35: Volume rendering from Scientific datasets at LLNL [23]. They are respectively: density, diffusivity, pressure and viscocity.

We compare our bi-Lorenzian L^2 predictor with other scanline predictors proposed for image compression: the Paeth predictor [81] used in the PNG image format [86], the median predictor used in JPEG-LS [105], and the L^1 Lorenzo predictor. All except L^2 predict a sample from the same set of three neighbors (Figure 12(a)).

In order to apply L^2 in a scanline traversal, two rows of previously coded samples must be maintained (Figure 37(a)). To bootstrap the predictor, one may use lower-dimensional Lorenzo prediction to first recover domain boundaries. Alternatively, one may use the spectral predictor for partially known neighborhoods.

Figure 36 shows the results of predicting multiple 2D slices of the single-precision floating point scalar fields shown in Figure 35 obtained from a fluid dynamics simulation [23]. On high-precision data like this, L^2 often offers substantially better prediction than predictors that use smaller stencils. The benefit of a larger stencil comes at the expense of higher sensitivity to quantization, due to accumulation of per-sample errors and larger (in magnitude) weights. Analysis shows that the prediction error due to quantization is three times larger for L^2 than for L^1. Hence L^2 generally performs worse than L^1 on low-precision

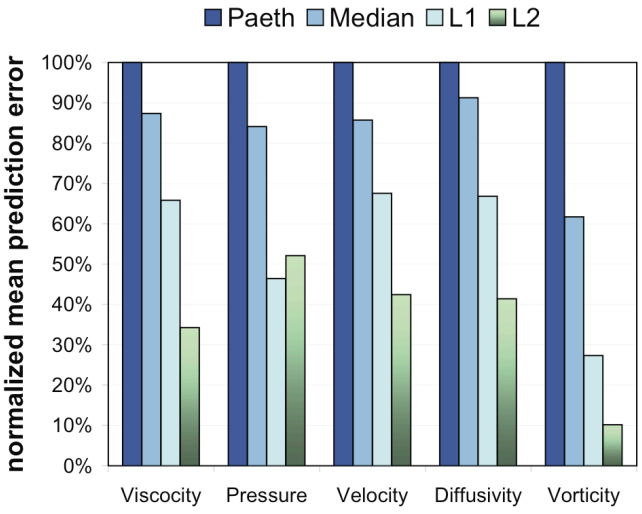


Figure 36: Scanline transmission comparison of predictors

data such as 8-bit images.

4.6.2 Progressive refinement

Often, datasets are transmitted progressively, doubling the resolution in x and y after each refinement. This is one of the possible hierarchical approaches discussed in Section 4.3.2. The missing values for a refinement may be transmitted in scan-line order, as show in Figure 37(b), which results in three 3×3 neighborhood configurations from which samples are predicted (Figure 37(c-e)).

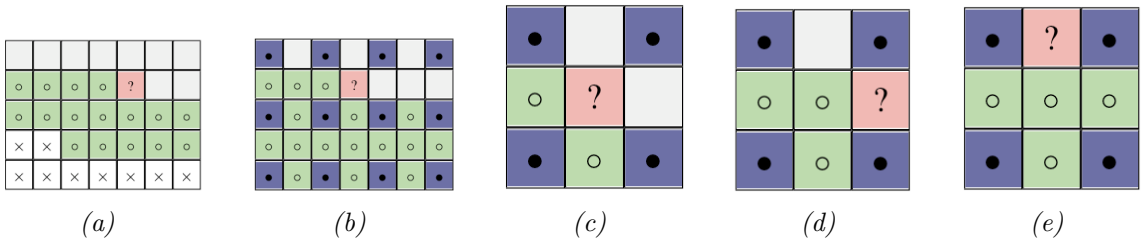


Figure 37: (a) L^2 footprint (circles) maintained during scanline traversal. (b) Coarse-resolution (solid) and fine-resolution (hollow) processed samples in a hierarchical traversal. Within each level of resolution, scanline traversal is used, resulting in three predictor stencils: (c) face, (d) vertical edge, and (e) horizontal edge sample.

We consider three predictors for the face sample (Figure 37(c)): bilinear interpolation H^1 of corner samples (Figure 12(d)), spectral prediction S_f (Figure 12(f)), and a hybrid

predictor H (Figure 12(e)) that first linearly interpolates the unknown neighbors at the vertical and horizontal edges from their immediate neighbors to fill the neighborhood and then predicts the face point using radial prediction R.

Note that both H^1 and H are special spectral predictors that simply ignore some of the known neighbors. For the edge points, H^1 and H resort to linear interpolation of corner points for prediction (since no other reasonable non-spectral predictor is available), while our spectral predictor is able to make use of all decoded samples (Figures 12(g) and 12(h)).

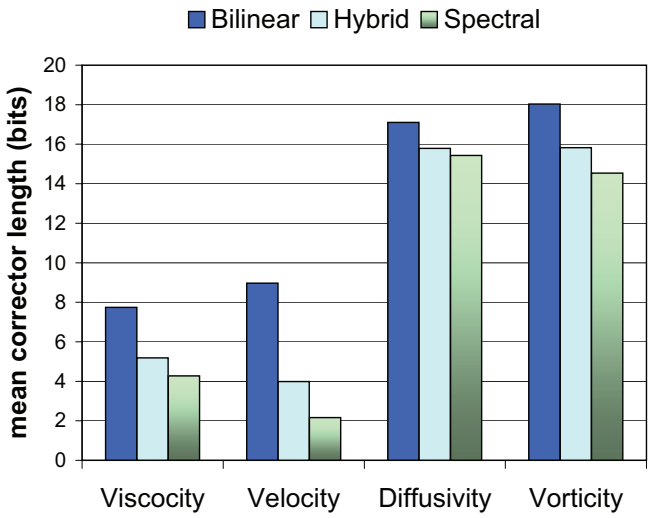


Figure 38: Comparison of predictors using a hierarchial progressive approach.

Figure 38 illustrates the advantage of using all known neighboring samples in the prediction. S_f offers in all cases superior prediction over H^1 and H, leading in one case to as much as a $4:1$ improvement in compression. Note that one may choose a different traversal order within each level. In fact, our experiments show that transmitting the missing edge samples first and then the face samples further improves compression, in part because the face samples may be predicted using the radial predictor with fully known (not simply estimated) neighborhoods.

4.6.3 Isocontouring

In many scientific, engineering, and medical applications, regularly sampled volumetric scalar fields are visualized in terms of isosurfaces. For instance, a remote viewer may wish

to see the isosurface $S(t)$ formed by all points at temperature t or to explore the family $S(T)$ of iso-surfaces with temperatures in a range $T = [t_{min}, t_{max}]$. Instead of transmitting the geometry of $S(t)$ or some compressed form of the animation $S(t)$, it is often more effective to transmit the minimal subset of scalar values needed to reconstruct the single iso-surface $S(t)$ or family of isosurfaces $S(T)$ [75]. To satisfy this query, one needs to transmit not only the samples with values in T, but also some of their neighbors to obtain a complete "scaffold" around the surface. In a scenario where the remote user decides to extend T to a larger interval, compression and incremental transmission of the subset of additional samples would often be preferable over complete retransmission. for both initial and incremental transmission, it is not obvious how to predict the irregular subset of sample values using traditional means.

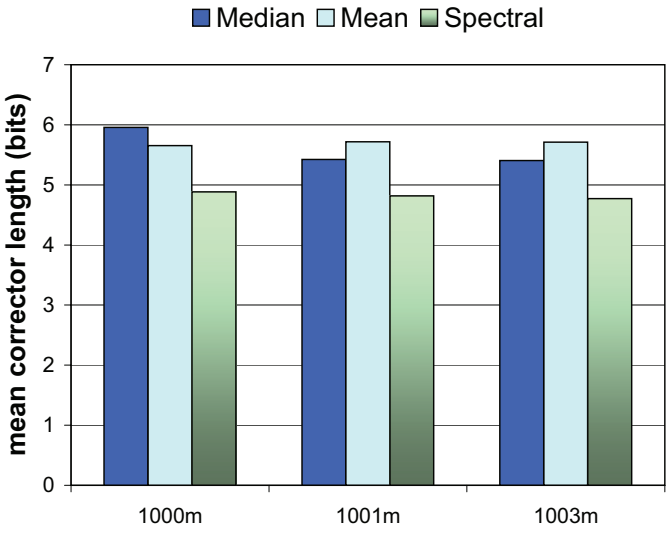


Figure 39: Comparison of predictors on compression an isocontour. It shows the versatility of the Spectral prediction.

Because we are only interested here in illustrating the benefits of the Spectral predictors, we will not discuss the transmission order nor how to encode the mask that identifies the missing samples. We focus on the prediction of the missing values and report experiments in 2D from the Puget Sound 16-bit terrain surface (available at `http:`

//www.cc.gatech.edu/projects/large_models/ps.html. We first extracted an isocontour at 1000m elevation and predicted all necessary samples, then incrementally transmitted missing vaues for isocontours at 1001m and 1003m, resulting in an average number of known neighbors of 5.13, 5.42 and 5.41 respectively. Since samples are often not available for predictors like L^1 to be applied, we compare our spectral predictor with predictions based on the mean and median sample value in a 3×3 neighborhood centered on the predicted sample. We observed consistent reduction in correct bit length (13-33%) using the spectral predictor, even for this lower precision data set (Figure 39).

4.7 Conclusion

We have proposed in this chapter the Lorenzo L^1 predictor, the bi-Lorenzo L^2, the radial R and Spectral predictors S, which are a family of optimal predictors for any stencil, and subsum the other three.

The Lorenzo predictor predicts the value of an n-dimensional scalar field F at a sample point v from its $2^n - 1$ previously processed neighbors that form the vertices of an n-dimensional hypercube. The predicted value for $F(v)$ is simply the weighted sum of all values of F at the other corners of the cube. The weights are either $+1$ or -1, depending on the minimal number of cube edges between the sample and v. The Lorenzo predictor is well suited for cases when a large dataset has to be compressed in its totality and there is coherence in nD. If the dataset is stored with high precision and is relatively smooth, we recommend the use of the bi-Lorenzo predictor.

We recommend a progressive hierarchical scheme using the radial predictor together with spectral predictors to compress a dataset where approximations are required. Our framework, which is based on the eigenstructure of the combinatorial graph Laplacian, while applied only to 3×3 neighborhoods in 2D regular grids here, easily generalizes to higher dimensions and to irregular grids. The drawback of spectral predictors is that computing the weighs in execution time (involving matrix inversions) is not feasible. Storing the weights in a table solves the problem if the number of weights to encode are few; yet the fact that the number of weights to store grows exponentially with the size of the neighborhood bars the

use of Spectral predictors in large neighborhoods, such as a 3D one. One possible solution is the selection of a subset of spectral predictors, exchanging versatility for space. Another solution is the reduction of the number of weights to store through the removal of redundant set of weighs using symmetry. This last solution diminishes the cost of storing the table; yet even with a drastic reduction the total size of the table exceeds usable size.

The Lorenzo predictor is exact for all polynomials of degree less than n, and its accuracy increases with the smoothness of the data. Because of the limited size of its footprint, the predictor is well suited for out-of-core streaming compression and decompression.

We argue that S is the best predictor for a 3×3 neighborhood; we provide a strategy for selecting the most promising neighborhood that contains f, and demonstrate the benefits of S over competing predictors in three simple applications. The benefits of the Spectral predictor can be extended to any neighborhood.

CHAPTER V

MESHES

Meshes, as seen in Chapter 2, are defined as structures that hold graphical data. Meshes represent solids in 3D by storing their surface, they are 2D surfaces embedded in 3D. In our work, meshes are represented as two tables: a connectivity table and a geometry table.

If we consider only the vertex coordinates, the storage cost before compression for the vertex information in a mesh is:

$$\text{Geometry Cost} = 32 * 3 * V \tag{14}$$

where 32 is the number of bits to store a floating point number, 3 is the dimension of the space, and V is the number of vertices. The storage cost for the connectivity is

$$\text{Connectivity Cost} = 32 * 3 * T \tag{15}$$

where 32 are the bits to store an integer reference, 3 is the number of vertices per triangle, and T is the number of triangles. In practice, the number of triangles is roughly twice the number of vertices; uncompressed connectivity data thus consumes twice more storage than vertex coordinates.

In this chapter we first explain techniques for mesh compression that focus on connectivity compression. This thesis describes techniques for predictive compression, which apply to geometry compression; thus we require connectivity processing techniques. Later we will describe methods for the simplification of meshes, a technique that computes an approximation of the original mesh with smaller connectivity. After simplification we explain progressive meshes, which are techniques that couple compression, simplification and refinements. The chapter ends with a description of geomorphs, used to create a seamless transition between meshes at different levels of simplification.

5.1 Mesh Compression

Due to the fact that connectivity requires more storage than geometry, there has been much work done in the area of connectivity compression. As stated by Gumhold [38]:

> Denny and Sohler [26] showed that for sufficiently large triangle meshes the connectivity can be encoded in a permutation of its vertices alone. This would make all work on connectivity coding useless. But there is a catch in it. The connectivity can be exploited to encode the vertex locations more efficiently. With a simple delta coding technique the connectivity information improves vertex locations encoding by about the amount of the storage space consumed by a permutation of the vertices, which grows with $O(vlbv)$ The connectivity itself only consumes $O(v)$ bits, what justifies its encoding.

Motivated by the need in rendering to use triangle strips, several connectivity compression methods follow a triangle strip or similar traversal in their compression.

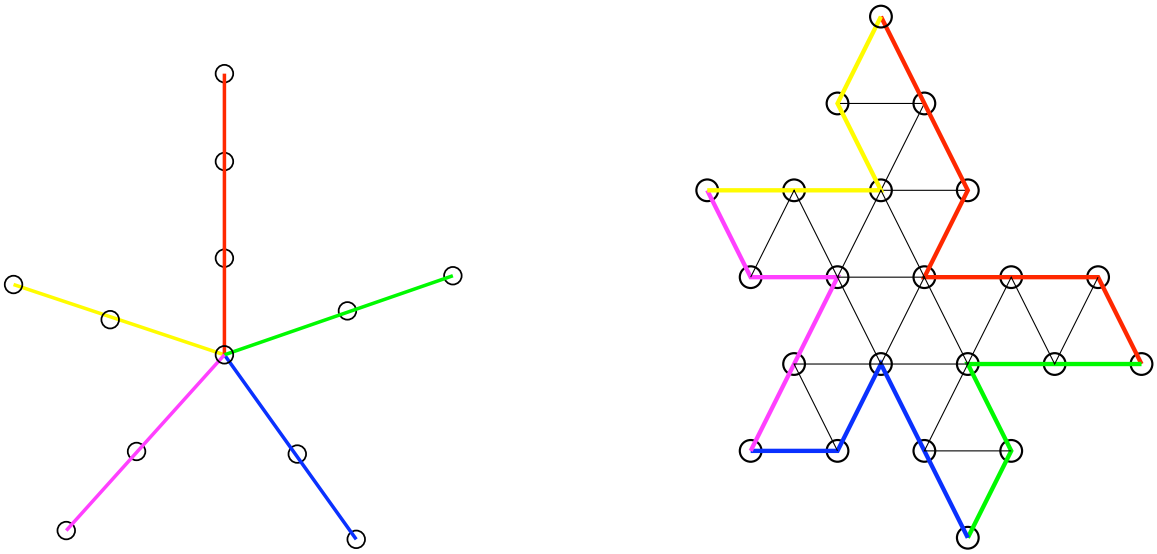


Figure 40: Structures used in the processing of a Dodecahedron on the approach proposed by Taubin and Rossignac [99]. On the left there is the vertex spanning tree that defines the triangle cutting of the mesh represented on the right.

Deering [25], pioneer in the area of mesh compression, proposed a method that traverses the mesh in triangle strips. Triangle strips specify each vertex twice (in the best case), hence

Deering proposed to keep a buffer of 16 vertices and refer to a vertex either as new or as a 4-bit code identifying a vertex in the buffer. For geometry compression, Deering encodes the vector between the previous vertex in the traversal and the vertex to be encoded, using variable length coding.

Taubin and Rossignac [99] developed *Topological Surgery*, a mesh compression method that cuts the mesh using a vertex spanning tree (see Figure 40). The result is encoded as a triangle spanning tree with the help of the previous vertex spanning tree. The method achieves an overall connectivity cost of four bits per vertex, including the encoding of both spanning trees. To encode the geometry, Taubin and Rossignac proposed predictive coding, where the previously decoded vertices are used to predict the next vertex, the correction is encoded. For each model, a set of optimal weights is computed to achieve optimal prediction accuracy. The weighs are used on the previously seen vertices; geometry compression totals about 14 bits per vertex.

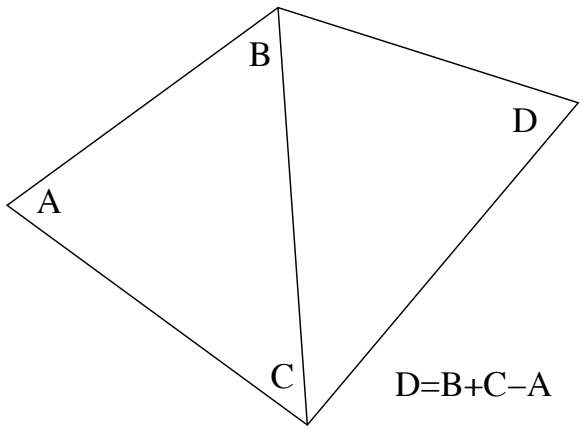


Figure 41: Parallelogram predictor introduced by Touma and Gotsman [101]. A plane is fitted to the ABC triangle; D is computed as $D = B + C - A$.

Touma and Gotsman [101] developed a method of mesh compression through a traversal similar to Taubin and Rossignac's. Touma and Gotsman differentiate between the encoding of two types of triangles, the ones that introduce a new vertex and the ones that split the border of the current region into two parts. Touma and Gotsman proposed a run length encoding scheme of the degrees for each vertex (valence coding), achieving an average of 2 bits per vertex for the very favorable cases. More importantly, in this paper they

introduced the *parallelogram predictor* (Figure 41), which is the specific instance of the Lorenzo predictor in 2D. The parallelogram predictor uses the previous triangle to predict the position of the vertex at the opposite triangle.

The sampling (positions of vertices) in triangle meshes is dependent on the curvature of the mesh (for most meshes). Flat areas have few vertices, regions with high curvature have a large number of vertices, implying a large number of triangles. This adaptive refinement is the cause of the compression rates achieved by the parallelogram predictor. The parallelogram predictor uses a plane for prediction. The difference between the plane and the real surface is dependent on the curvature and the distance between samples. In meshes where the distance between samples (the sampling density) is dependent on the curvature, the corrections of the parallelogram predictor are highly biased (not towards zero), thus achieving good compression rates. The parallelogram has been extended and used in innumerable articles. This method was improved by Alliez and Desbrun [4], who proposed a variation in the traversal, where vertices with more decoded triangles are given preference. Such heuristic in the traversal reduces the number of splits, improving the entropy of the symbols used to encode the types of triangles. Alliez and Desbrun's method reduces the number of splits in the mesh at least by 50% in most meshes. For large regular meshes with no splits, Alliez and Desbrun achieved a total upper bound of 3.24 bits per vertex.

Rossignac improved upon Topological Surgery with Edgebreaker [88]. This work, acclaimed by the community with several awards, compresses a triangle mesh in a growing fashion. A triangle that adds a new vertex is labeled a C, if the triangle has the left edge already used, it is labeled an R, and if it is the right edge used, it is labeled L. If the triangle splits the border of the current region, it is label an S, and if it is the last triangle in a region it is labeled an E (see Figure 42). This coding of triangles into symbols uniquely captures the connectivity of the mesh, and allows for triangles to be coded using 1.8 bits per triangle, or 3.67 bits per vertex as an upper bound, although it is common to achieve 0.8 bits per triangle. The encoding of the geometry follows Touma and Gostman's method [101] using the parallelogram predictor. This work has been extended to support meshes with handles [70], and in several other publications introducing different traversals and improved

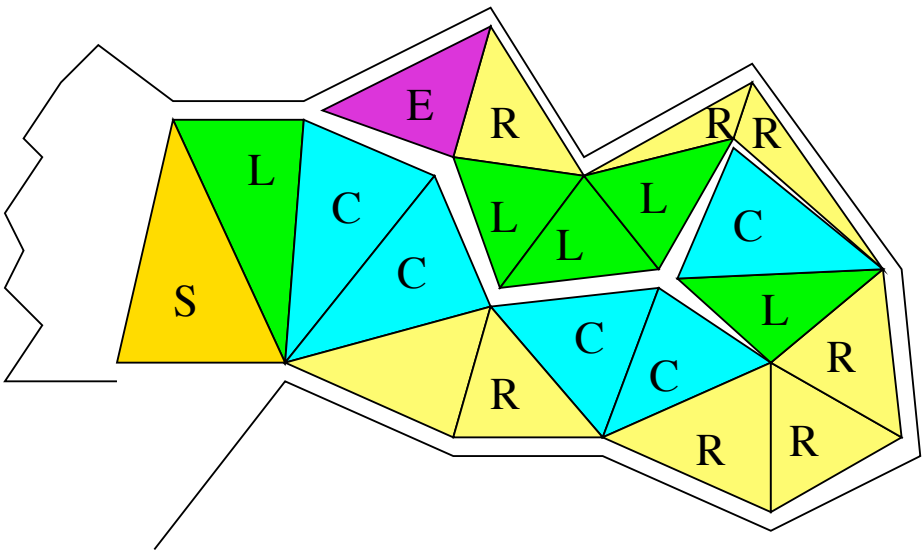


Figure 42: Representation of the Edgebreaker [88] traversal. Triangles with different symbols from the {C,L,E,R,S} string are colored differently.

symbol encodings [54, 89, 90].

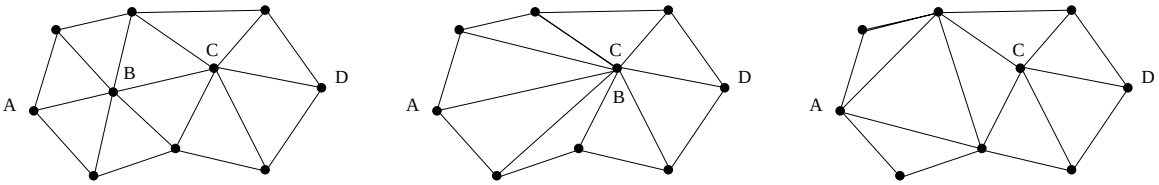
Gumhold's Cut-Border Machine [37] is also a mesh compression algorithm based on growing an area on the triangle mesh. The approach was developed independently with Edgebreaker. The Cut-Border Machine is very similar to Edgebreaker, but has more symbols than the {C,L,E,R,S} string, such as the O symbol used to encode a border edge.

Triangle meshes where all triangles have valence 6 achieve greater compression than meshes with a wide distribution of valences. Remeshing a mesh consists in generating a new connectivity and geometry sets that approximate the original mesh while achieving a distribution of valences biased towards six. Kodakovsky et al's method [58] and Attene et al's SwingWrapper [8] are contributions in this area. The focus of this thesis is not in remeshing; therefore we do not pursue this further.

5.2 Mesh Simplification

Mesh simplification algorithms reduce storage size of the mesh by collapsing one edge at a time or discarding a vertex and creating new triangles to fill the hole left by the vertex [20, 59]. *Vertex decimation* consists in removing a vertex from geometry and connectivity. The triangles incident upon the vertex are also removed, creating a hole in the mesh. The

last step in vertex decimation consists in adding triangles to fill the hole. In Figure 43(c) we depict an example triangulation after the decimation of vertex B. *Edge collapse* (see Figure 43(b)) has been used by Hoppe and Ronfard [44, 46, 87], amongst others. An edge collapse consists in merging two adjacent vertices (B and C in Figure 43). As a consequence, two triangles are reduced to a single edge, and the collapsed edge disappears. Edge collapse is defined by the collapsed edge and the final position of the merged two vertices. To select the edge to collapse, all edges are assigned an error value, quantifying how much the collapse of that edge would distort the mesh. A triangle mesh is composed of planar triangles; the mesh approximates a surface by the union of small planar polygons. Through simplification, the positions of vertices change, triangles are removed and triangles change their plane. A common estimate for the error associated with each vertex of the mesh is its distance to the set of planes defined by the triangles incident upon the vertex. On each simplification step, the edge with the smallest estimated error is collapsed, and the edges incident upon the vertices merged have their error value recomputed.



(a) original *(b) edge collapse* *(c) vertex removal*

Figure 43: A portion of a triangle mesh is depicted on (a), the resulting connectivity of collapsing the BC edge is depicted on (b), the resulting connectivity of removing vertex B is depicted on (c).

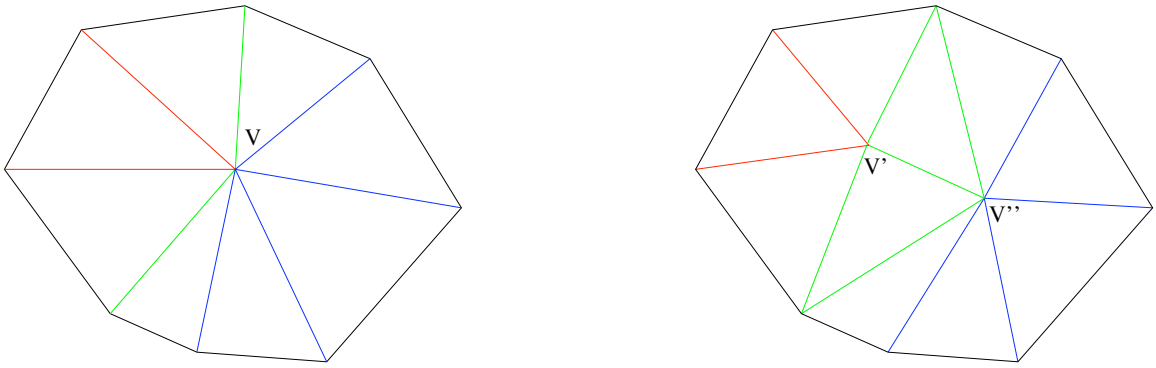
Successive edge collapses result in the merging of several vertices, defining vertex clusters. On a simplified mesh, each vertex has an associated vertex cluster, with one or more vertices, and with the planes of all the triangles incident upon a vertex in the cluster. Already collapsed triangles also have their planes associated with the cluster. The cluster and the set of planes are used to estimate future collapse errors and the current deviation of the simplified mesh from the original mesh.

Ronfard and Rossignac [87] proposed to estimate the simplification error as the maximum distance from a vertex to the planes adjacent on it. Ronfard and Rossignac's technique sets an upper bound on the error introduced by simplification in any position of the mesh, L_∞.

Garland and Heckbert [33] proposed a method to estimate the error introduced when an edge is removed by using a 4×4 quadratic matrix for each vertex in the mesh. Computation and management of the error matrices consist only in the addition of coefficients when two vertices merge. As such, the error estimate Garland and Heckbert compute is the sum of the distances from the vertex to each plane. As a result, the mesh is simplified minimizing the quadratic error L_2.

5.3 Progressive Meshes

Reversing the simplification process is harder because the discarded information has to be transmitted. The reverse of an edge collapse is a *vertex split*. A vertex split is defined by the vertex to split, and two incident edges. In Figure 44(a) there is a mesh. The vertex split of vertex V is defined by marking V and by marking two edges, which in this figure are colored green. The result of the split are two vertices, V' and V'', and two new triangles formed by the split of the green edges. The position of the two new vertex locations has to be encoded or estimated based on the local geometry.



(a) original *(b) vertex split*

Figure 44: A vertex split is defined by the vertex V, and by two incident edges (colored in green). After the split, V becomes two new vertices, and the green edges duplicate creating two new triangles.

Progressive mesh compression strives to compress a triangle mesh in such a way that if the decompression were to be stopped at some point, for example due to a time out, the recovered data would have the smallest error estimation of the final mesh. Progressive mesh techniques encode a coarse simplified version of the mesh followed by refinements until the original mesh is recovered. The refinements are ordered based on the error the correspondent simplification introduced.

Hoppe's [44] pioneering work on progressive meshes, was not focused on compression. He was the pioneer in edge collapse and vertex split, but the default encoding of a vertex split cost him $(\log(n)+5)n$ bits. The $\log(n)$ cost comes from identifying 1 (out of n possible vertices) vertex to split in each situation, and the 5 bits come from identifying the two green edges.

Several progressive compression methods use batches of data to increase performance. Bajaj et al [11] proposed to divide the mesh into layers, and process the layers with inter-layer simplification and refining through vertex removal, and intra-layer simplification, where the simplification process alters the topology of the mesh. Bajaj reduced the cost of encoding the vertex split to linear instead of Hoppe's $n \log n$ using the locality property of the layering structure. The reduction in cost comes at the expense of having to encode as many simplification information per layer as possible, thus having less control on the error approximation for the mesh.

Pajarola and Rossignac [82] devised a method to encode the progressive refinements. They transmitted the refinements in batches, thus minimizing the cost of indicating each vertex split. With each encoded batch the technique increases the number of vertices by up to 50 %; the amortized cost per triangle of 4 bits also allows to indicate how the topological refinements should be applied. The new positions of the vertices are estimated using as prediction a reverse variant of the edge-split Butterfly subdivision scheme. The Butterfly subdivision scheme computes the position of a new vertex as a weighted sum of the surrounding vertices.

Alliez and Desbrun [3] based their progressive scheme on vertex removal, but instead of

removing the vertex with the lowest error impact, they removed vertices based on their valence. Optimal error removal produces almost random vertex removal order, and that costs several bits to encode. By decimating vertices with a known valence range (between four and six), good compression is achieved, while striking a compromise in error introduction. Alliez and Desbrun claim to have achieved an average of 3.69 bits per vertex.

5.4 Geomorphs

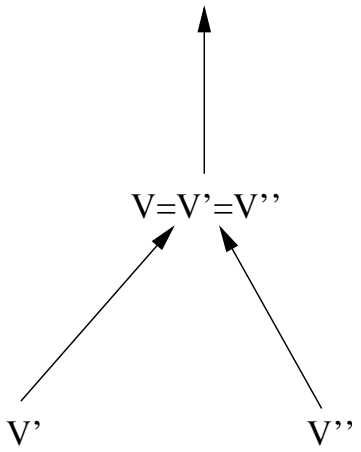


Figure 45: Part of the directed cluster tree mapping that defines a geomorph.

When updating the mesh, the batches of data result in a large increase in geometric detail, which can be seen as a popping and annoying effect. To mitigate this issue, it is common to see the progressive refinement coupled with geomorphs during rendering. A geomorph allows the smooth visual transition between any two meshes. The geomorph is essentially a copy of the first mesh, but whose attributes at vertices and edges interpolate between the first mesh and the second. Referring to Figure 44, a geomorph between the start and the refined mesh is a sequence of triangle meshes with the connectivity of the refined mesh. The sequence starts with vertices V' and V'' at the same location as the location of vertex V from the original mesh. The sequence progresses by updating the positions of V' and V'' until the vertices are in the correct position for the refined mesh. As a result, the visual effect of the sequence of meshes is that a triangle which previously was appearing suddenly, now grows smoothly on the mesh. To be able to perform a geomorph,

the correspondence between vertices of the start mesh and the end mesh is needed; the correspondence defines the path that vertices traverse on the geomorph. Such correspondence can be seen as a tree, see Figure 45.

In the case of Hoppe [44], the mapping between vertices at different levels of simplification is well established. This relationship enables the use of a geomorph. In cases where this mapping between the two meshes does not exist, for example in sequences of isosets, it is necessary to approximate the mapping in order to be able to perform a geomorph. The alternative to using geomorphs is to transition the meshes using alpha blending. The use of alpha blending may lead to ghost effects when a part of the mesh is disappearing and another part is appearing, or it may create the effect of blurry images.

CHAPTER VI

ANIMATED MESHES

The entertainment industry was revolutionized when the process of displaying images in fast succession, the video, was invented. Animated meshes follow the same basic principle. On a videogame, when a character appears to be moving, it is displaying as several renderings of of several poses of the character in fast succession. Object deformations, even explosions fall into the category of animated meshes. In this chapter we will propose methods for the lossless and lossy compression of animation. We work with several animations, depicted on Figures 46, 47 and 54. The work introduced in this chapter was published in part in [48].

6.1 Introduction

Although animated 3D models may be produced and represented in a variety of ways [43], they are often stored and transmitted as series of consecutive frames, each represented by a triangle mesh. In general, the connectivity of the triangle mesh and even the topology of the surface it represents may evolve with time. Nevertheless, we restrict our attention to a reasonably large class of animations in which the connectivity is *identical* in all frames, similarly to several recent pioneering efforts [64] in animation compression. This class comprises many physic-based animations of cloth deformations [100] and animations produced by warping space [94] [12] [69].

We assume that the mesh contains T triangles and V vertices and that the animation has F frames.

In a sequence of meshes with constant connectivity, the cost of encoding the geometry, the positions of the vertices at each frame, far outweighs the cost of encoding the connectivity, which is done once per animation. Consequently, we focus on the compression of the geometry, which basically amounts to the compression of the three coordinates of each vertex, $\mathbf{v}$, in the successive frames of the animation.

It is possible to encode each mesh on its own, using methods described in Chapter 5.

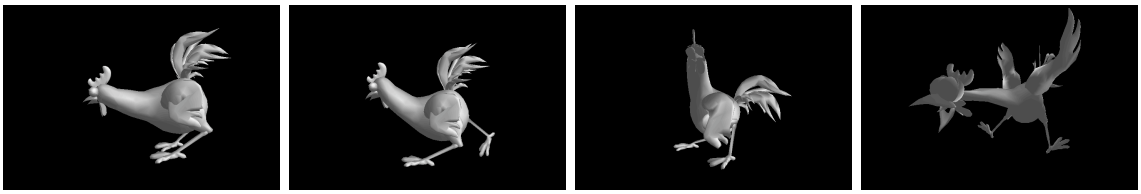


Figure 46: Chicken animation, courtesy of Lengyel, (c) by Microsoft. This animation has become a standard in compression, it has been used as a benchmark in several papers. The mesh has 3030 vertices and 5664 triangles.

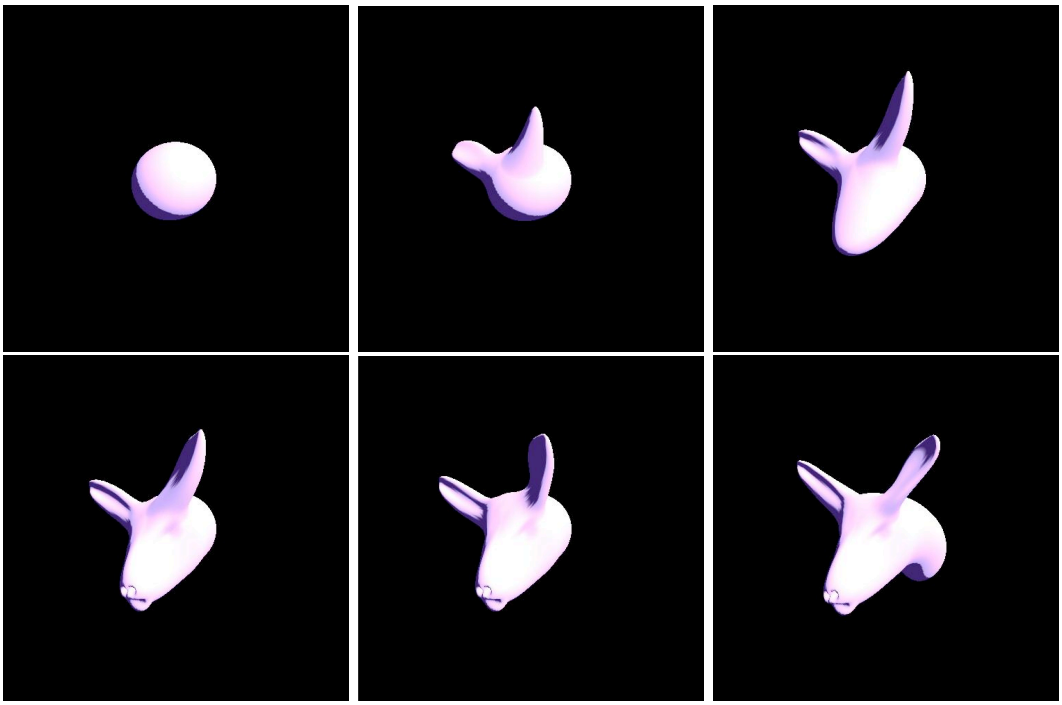


Figure 47: Kangaroo head model build with Twister, courtesy of Ignacio Llamas. The figure represents the animation that deforms a sphere into a kangaroo head.

But doing this would be the same as applying a 2D prediction to a 3D dataset; which we discussed in Chapter 4 and concluded not optimal.

The original animation with constant connectivity is specified by a connectivity graph G and a series of tables V_j containing the vertex-locations for frame j.

6.2 Prior art in animated meshes

Several triangle mesh compression algorithms were discussed in Chapter 5. Several relatively recent efforts have pioneered the space-time compression of 3D animations of freely deforming surfaces, as opposite to rigid body motions or to physically plausible simulations

of articulated bodies.

Lengyel [64] proposed several fitting predictors to compress the motion of the vertices of animated triangle meshes of a constant connectivity. His method divides the vertices of the mesh into groups and computes a transformation that best matches the average evolution of the vertices in each group. The types of transformations it can fit include Affine Transformations, Free-Form Deformations, Key-Shapes, Weighted Trajectories and Skinning. All other deformations are approximated by one of those. Then it encodes the differences (residues) between the real position of each vertex in each frame and the position predicted by applying the corresponding transformation. When large portions of the model are subject to perfect instances of these transformations, the approach is extremely effective. But the optimal partitioning of the mesh and the fitting of good transformations remains a delicate and computing intensive task. Instead of attempting to generate optimal partitions and optimal transformations, Lengyel proposes a simpler, sub-optimal approach. It selects a subset of the triangles and, for each one for these triangles, computes the transformation that interpolate the evolution of their geometry through the desired frames. Triangles undergoing similar transformations may be merged. Then, other vertices are associated with the triangle whose motion best matches theirs.

Alexa and Müller [2] proposed an interpolation predictor. They start by normalizing the animation. To do so, they translate all frames so that the origin lies at the center of mass of the model. Then they apply an affine transformation to it minimizing the sum of the square of the displacement for each vertex with respect to the initial frame. Put together, those modified frames form a large matrix, having for dimension the number of frames and three times the number of vertices in the mesh. Using an expensive PCA (Principal Component Analysis), they compute the eigen values of the product of that matrix with its transpose. These define eigen vectors and a coordinate system, whose axes are aligned with the principal components of the deformation. Thus the deformation is represented by this change of coordinate systems and by its coefficients in it. By setting to zero most of these coefficient, except the largest ones, they produce an approximating animation, which may be encoded with fewer bits. By sending more coefficients, they progressively refine the

animation.

Shamir and Pascucci [95] proposed a lossy animated triangle mesh compression approach. They introduced the *TDAG* structure (Temporal Directed Acyclic Graph) that holds spatial and temporal information about the mesh. Through the TDAG, the authors are able to create approximations of the animation through spatial (simplification) and temporal (frame sampling) decimations. The purpose of the TDAG is not compression, its authors discuss it in the context of approximation computation purposes only.

The recent work of Karni and Gotsman [57] follows along the same lines as Alexa and Muller's, it improves the compression of the PCA transformation using linear prediction coding. The method works at its best when the animation is quite large, and each individual frame is coarse, to amortize the cost of storing the eigenmodes. It is not suited for animations with large number of vertices.

Al-Regib et al. [1] propose a combined approach where a possibly different set of key vertices is selected in each key-frame. Their trajectory is encoded along as they retain the status of key vertices. The trajectories of other vertices are estimated through interpolation of these key vertices.

Guskov and Khodakovsky [39] presented a wavelet approach to compress an animation. It computes a first parametric model, builds a progressive model out of it, and uses the model to guide the wavelet decomposition of each frame in the animation. It works well for animations that keep the same parametrization over all the sequence, but when that is not the case, the progressive model encodes vertices that are not the most significant at that time in a particular frame.

Briceño et al [15] proposed the Geometric Video approach, that converts every frame of the animation into a geometric image [36]. Finding the correct cut of a geometric image for all frames is hard, what works best on a frame of the animation is not shared on subsequent frames. Also, for coarser meshes with creases, special care needs to be taken.

Carr and Hart [18] have proposed a method for fast reclustering in dynamic meshes. Their approach is not targeted for simplification, yet it could be adapted to compute a subobtimal simplification by evolving the simplification of a previous frame. We did not

integrate their idea into our approach because we consider all frames at the same time. Nonetheless, their approach suggests several ideas that we wish to pursue in the future work.

In order to ensure an apparent continuity in the behavior of the animated shape, most animations are finely sampled in time, and hence exhibit a significant amount of temporal coherence, which is untapped if the frames are compressed independently of each other. Inspired by this observation, one may consider encoding the trajectory of each vertex independently. Because we need not only to encode the path followed by each vertex, but its position as a function of time, we can cast this problem as the compression of a curve in the four-dimensional space-time domain or as the problem of computing a concise representation of a parametric curve $\mathbf{v}(t)$. A variety of curve fitting approaches, reviewed in [91], could be considered here. We have decided not to pursue these trajectory compression approaches because they do not exploit the spatial coherence present in most animations.

6.3 The Dynapack Algorithm

Dynapack is our proposed method for the lossless compression of animated meshes with constant connectivity. To precisely describe the Dynapack algorithm, we first discuss in this section the data structure used to represent the connectivity of the triangle mesh, which is identical for all frames. We then explain the traversal of the mesh and its use to perform calls for the various predictors when encoding the vertex locations.

6.3.1 Corner table data structure and operators

We use the Corner Table [89] to store the connectivity that is common to all the frames and to traverse their vertices in an order suitable for the various predictors discussed here.

The **geometry** (i.e., vertex coordinates) is stored in the **coordinate table**, **G**, where G[*v*,*f*] contains the triplet of the coordinates of the location of vertex number *v*, in frame *f*. For conciseness, we denote it by v.g(f).

Triangle-vertex **incidence** defines each triangle by the three integer references to its vertices. These references are stored as **consecutive** integer entries in the **V table**. Note that each one of the $3T$ entries in **V** represents a **corner** (association of a triangle with

one of its vertices). Let the integer c define such a corner. (We will abuse the language and speak of corner c, rather than of the corner number c.) Let c.t denote its triangle and c.v its vertex. Remember that c.v and c.t are integers in $[0, V-1]$ and $[0, T-1]$ respectively. Let c.p and c.n refer to the previous and next corner in the cyclic order of vertices around c.t.

As discussed in Chapter 2, the **G** and **V** tables suffice to completely specify the triangles and thus the surface they represent. But they do not offer direct access to a neighboring triangle or vertex. We chose to use the reference to the **opposite** corner, c.o, which we cache in the **O table** to accelerate mesh traversal from one triangle to its neighbors. For convenience, we also introduce the operators c.l and c.r, which return the **left** and **right neighbors** of c (see Fig. 48).

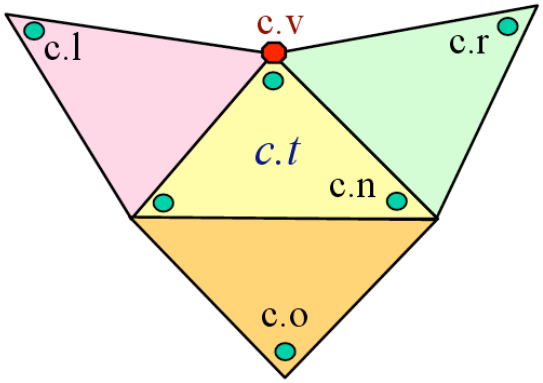


Figure 48: Corner operators for traversing a corner table representation of a triangle mesh.

Note that we do not need to cache c.t, c.n, c.p, c.l, or c.r, because they may be quickly evaluated as follows: c.t is the integer division c.t DIV 3; c.n is $c-2$, when c MOD 3 is 2, and $c+1$ otherwise; and c.p is c.n.n; c.l is c.n.o; and c.r is c.p.o. Thus, the storage of the connectivity is reduced to the two arrays, **O** and **V**, of integers.

We assume that all triangles have been consistently **oriented**, so that c.n.v=c.o.p.v for all corners c. Note that **V** may be trivially extracted from most formats for triangle meshes and that **O** may be efficiently recovered from **V** [89].

To discuss the traversal of the mesh, we use the Boolean c.t.m to indicate that triangle c.t has already been visited. Similarly, the Boolean c.v.m indicates that vertex c.v has

been visited. For each frame, except the first one, and for each connected component of these frames, we start by encoding the 3 vertices of a first triangle, c.t, using a time-only predictor. (The integer c may be arbitrarily chosen to be for example 0.) We mark this triangle and its vertices as visited. Then, we issue three calls: dynapack(c.o), dynapack(c.l), dynapack(c.r). These invoke the simple dynapack compression procedure presented below. It visits the other triangles of this component of the frame in a depth-first order of the triangle spanning tree and encodes its vertices. We use the convention which sets c.0 to -1 for corners that do not have an opposite corner, because their opposite edge is a border and has a single incident triangle. This can be seen in Listing 6.1.

Listing 6.1: Dynapack compression pseudocode.

```
dynapack(c) {          // frame's component compression
  if (c==-1) return; // return if a border is reached
  if (!c.t.m) {        // if triangle c.t is not yet visited
    if (!c.v.m) {      // if tip vertex is not yet visited
      encode(c.v.g(f) - predict(c,f)); // encode residue
      c.v.m = true;    // mark the tip vertex as visited
    }
    c.t.m = true;      // mark the triangle as visited
    dynapack(c.r);     // process the right neighbor
    dynapack(c.l);     // process the left neighbor
  }
}
```

Decompression follows the same pattern (see Listing 6.2). For each frame, except the first one, and for each connected component of these frames, it starts by decoding the 3 vertices of a first triangle, c.t, using a time-only predictor. It marks this triangle and its vertices as visited. Then, it issues three calls: dynaunpack(c.o), dynaunpack(c.l), dynaunpack(c.r), to the decompression procedure below.

The first frame is compressed and decompressed using a space only predictor. Subsequent frames can use predictors that use the previous mesh to achieve improved compression. We compare several predictors in the next section.

The coherence between neighboring vertices in meshes of finely tiled smooth surfaces reduces the average magnitude of the residues (i.e. of the coordinates of $\mathbf{c}$). Still, some of

Listing 6.2: Dynapack decompression pseudocode.

```
dynaunpack(c) {        // frame's component decompression
  if (c==-1) return; // return if a border is reached
  if (!c.t.m) {       // if triangle c.t is not yet visited
    if (!c.v.m) {     // if tip vertex is not yet visited
      c.v.g(f) =  predict(c,f) + decode(); //decode, add residue
      c.v.m = true;  // mark the tip vertex as visited
    }
    c.t.m = true;     // mark the triangle as visited
    dynaunpack(c.r); // process the right neighbor
    dynaunpack(c.l); // process the left neighbor
  }
}
```

the residues may be large. Thus, good prediction, by itself may not lead to compression. However, the distribution of the residues is usually biased towards zero, which makes them suitable for statistical compression [93]. Entropy or arithmetic compression is particularly effective if the coordinates or the residues are quantized to a small number of bits, typically ranging between 8 and 12. Such a quantization truncates the vertex coordinates to a desired accuracy and maps them into integers that can be represented with a limited number of bits. To do this, we first compute a tight (min-max), axis-aligned bounding box around the space swept by the model during animation. The minima and maxima of the x, y, and z coordinates, which define the box, will be encoded and transmitted with the compressed representation of he animation of each object. Then, given a desired accuracy, e, we transform each x coordinate into an integer $i = INT(\frac{x-xmin}{e(xmax-xmin)})$, which ranges between 0 and 2^B, where $B = \log_2(\frac{xmax-xmin}{e})$ is the maximum number of bits needed to represent the quantized coordinate i. The y and z coordinates are quantized similarly. In practice, for static meshes, the combination of the quantization, prediction, and statistical coding reduce the storage of vertex location data to between 3 and 9 bits per coordinate, depending on the quantization and the sampling rate of the surface relative to the size of its features.

6.4 Predictors for Meshes

We describe here four formulae for computing the extrapolating predictor, predict(c,f), from a selected set of the previously visited immediate neighbors of c.v in frames f and possibly $f-1$.

6.4.1 Space-only Predictor

The space-only predictor is used for encoding the first frame in Dynapack. For comparison, we also provide the result of using it to encode all the frames independently. It is exactly the parallelogram predictor popularized by Touma and Gotsman [101]. With this approach, predict(c,f) returns $c.n.v.g(f) + c.p.v.g(f) - c.o.v.g(f)$, as shown Fig. 49.

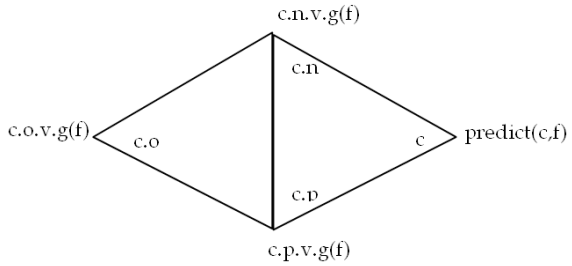


Figure 49: Space-only predictor: predict(c,f) = c.n.v.g(f)+ c.p.v.g(f)-c.o.v.g(f).

The space-only prediction amounts to fitting a plane, the same plane where the triangle used for prediction lives on. Triangle meshes are not flat, but they are sampled in an adaptive way, to capture the changes in curvature. The sampling makes that the error a linear prediction like a plane makes is almost constant for each triangle, and thus the entropy of the residues is very small. As a result, the space-only prediction, even though has less prediction accuracy than other more complex predictions achieves better compression, it has become a de-facto prediction.

6.4.2 Time-only Predictor

We have also implemented a time-only predictor, which does not exploit any spatial coherence and simply returns c.n.v.g($f-1$), which is the position occupied by the vertex in the previous frame. This time-only predictor has also been used by Lengyel [64] as a "row Predictor" and is a special case of Linear Predictive Coding.

6.4.3 Space-time Extended Lorenzo Predictor (ELP)

The first space-time predictor, proposed here, is a generalization of the Lorenzo predictor (See Chapter 4) developed for compressing regular samplings of four-dimensional scalar fields. The proposed generalization simply evaluates predict(c,f) as $c.n.v.g(f)+c.p.v.g(f)-c.o.v.g(f)+c.v.g(f-1)-c.n.v.g(f-1)-c.p.v.g(f-1)+c.o.v.g(f-1)$, as illustrated in Figure 50. This is the adaptation of the 3D instance of the Lorenzo predictor.

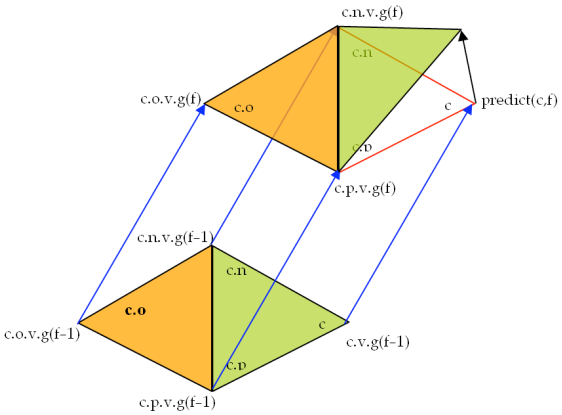


Figure 50: ELP: $predict(c,f) = c.n.v.g(f) + c.p.v.g(f) - c.o.v.g(f) + c.v.g(f-1) - c.n.v.g(f-1) - c.p.v.g(f-1) + c.o.v.g(f-1)$.

Note that ELP predicts perfectly the locations of the vertices of regions of the mesh that have been transformed by a pure translation from the previous frame. Indeed, if for all corners c, $c.v.g(f) = c.v.g(f-1) + \mathbf{d}$, then $predict(c,f) = c.v.g(f-1) + \mathbf{d}$. Thus, the residue from the prediction is zero.

6.4.4 Space-time Replica Predictor

To make our predictor capable of perfectly predicting rigid body motions and uniform scaling transformations, we have developed the new Replica predictor. It computes the coefficients a, b, and c, such that the vertex, $c.v.g(f-1)$, can be written as $c.o.v.g(f-1)+aA+bB+cC$, with $A = c.p.v.g(f-1) - c.o.v.g(f-1))$, $B = c.n.v.g(f-1) - c.o.v.g(f-1)$, and $C = \frac{A \times B}{\sqrt{\|A \times B\|}}$.

To compute a, b and c, we define $\mathbf{D} = c.v.g(f-1) - c.o.v.g(f-1)$ and write:

$$D = aA + bB + cC \tag{16}$$

Given that C is orthogonal to A and B, a dot product of both terms of the previous equation with vector A yields:

$$A \cdot D = aA \cdot A + bA \cdot B \tag{17}$$

Dot product with B yields:

$$B \cdot D = aB \cdot A + bB \cdot B \tag{18}$$

Solving the system of linear equations defined by equations 16, 17 and 18 yields:

$$a = \frac{A \cdot D * B \cdot B - B \cdot D * A \cdot B}{A \cdot A * B \cdot B - A \cdot B * A \cdot B} \tag{19}$$

$$b = \frac{A \cdot D * A \cdot B - B \cdot \ D * A \cdot A}{A \cdot B * A \cdot B - B \cdot B * A \cdot A} \tag{20}$$

$$c = D \cdot \frac{A \times B}{\|A \times B\|} * \sqrt{\|A \times B\|} \tag{21}$$

Where a, b and c are scalars, and $*$ represents the scalar product. Then, the vertex $c.v.g(f)$ is predicted by $predict(c, f) = c.o.v.g(f) + aA' + bB' + cC'$, where $A' = c.p.v.g(f) - c.o.v.g(f))$, $B' = c.n.v.g(f) - c.o.v.g(f)$, and $C' = \frac{A' \times B'}{\sqrt{\|A' \times B'\|}}$. The situation is illustrated Fig. 51.

Less formally, the Replica predictor looks at the previous frame and expresses vertex $c.v.g(f-1)$ as in a coordinate system derived from triangle ($c.o.v.g(f-1)$, $c.n.v.g(f-1)$, $c.p.v.g(f-1)$). More precisely, we compute the projection of $c.v.g(f-1)$ onto the plane supporting the previous triangle ($c.o.v.g(f-1)$, $c.n.v.g(f-1)$, $c.p.v.g(f-1)$). Then we compute the distance from $c.v.g(f-1)$ to that plane and encode a function of its ratio, c, to the area of the adjacent triangle. The function we use guarantees that the replica predictor will not be affected by a change of units. Furthermore, we compute two coefficients, a and

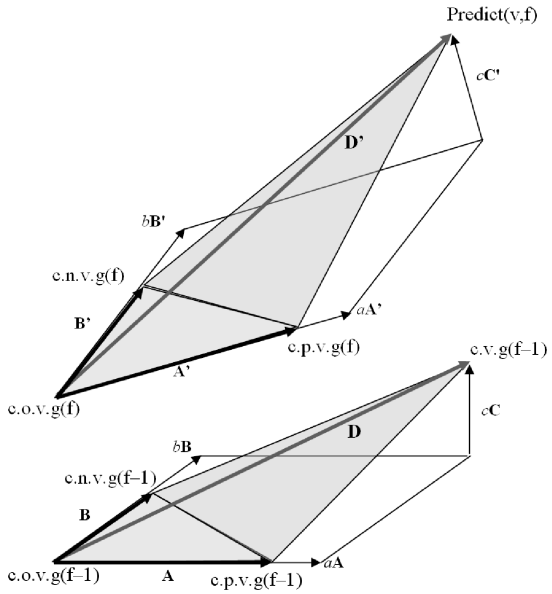


Figure 51: Replica Predictor.

b, such that the vector between $c.o.v.g(f-1)$, and $c.v.g(f-1)$ may be expressed as $aA+bB$, where A and B are the vectors joining $c.o.v.g(f-1)$ to the other vertices of the adjacent triangle. Thus, $c.o.v.g(f-1)$ and $c.v.g(f-1)$ are the opposite corner of a parallelogram with sides parallel to A and B.

Then, we replicate this construction on frame f, using the a, b and c, and coefficients computed from frame $f-1$, to estimate $c.v.g(f)$. Because this reconstruction only depends on the position of the previously visited triangle in frame f, the Replica is a perfect predictor when both triangles in frame f were obtained by moving the corresponding triangles in frame $f-1$ by the same rigid body motion. Furthermore, as we pointed out earlier, we have chosen the coefficient c to ensure that the predictor is independent of the chosen units and this will also be a perfect predictor if frame f is obtained by a rigid body motion and uniform scaling from frame $f-1$.

Clearly, Replica predicts perfectly the locations of the vertices of regions of the mesh that have been transformed by a rigid body motion, because the construction is relative to the neighboring triangle and thus is not affected by a rigid body transformation.

The normalization of C, dividing $A \times B$ by $\sqrt{\|A \times B\|}$ was introduced to ensure that Replica is also a perfect predictor for uniform scaling. In uniform scaling, vectors A and

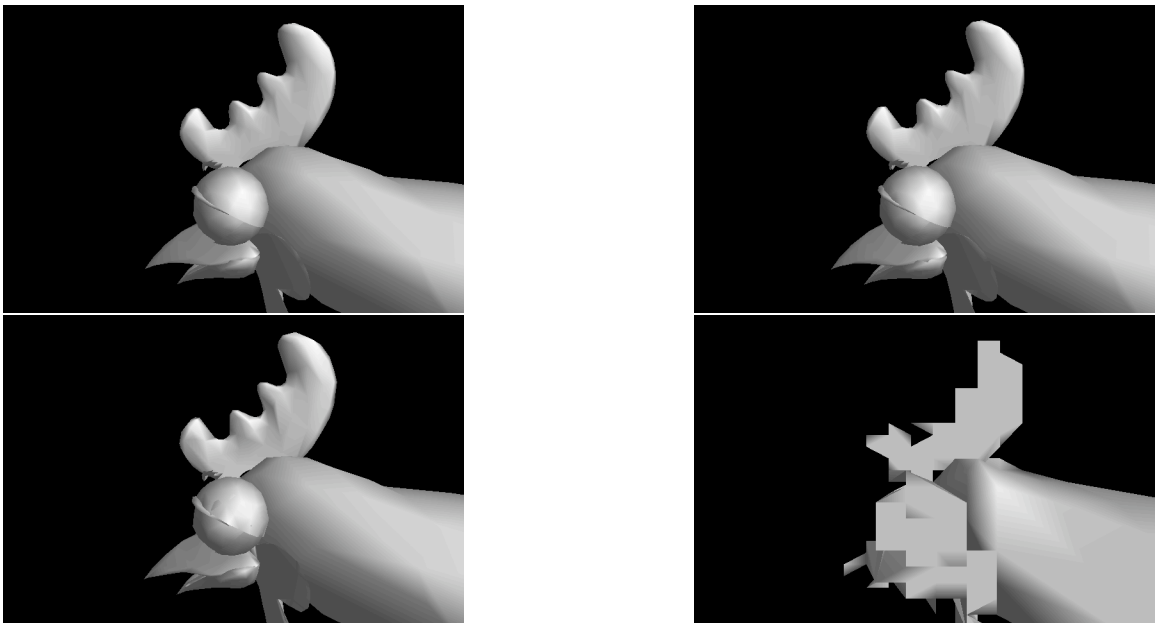


Figure 52: Chicken head at different quantization levels: full precision (up left), 13-bit quantization (up-right), 11 bit quantization (down-left) and 7-bit quantization (down-right).

B are scaled by the same factor s. Therefore, the cross product of the scaled versions of A and B is scaled s^2 with respect to the cross product of A and B. The division by the square root reduces the scaling factor back to s.

6.5 Lossless Animation Compression Results

To demonstrate the effectiveness of our two time-space predictors, and to compare them to the time-only and space-only predictors and to results obtained by others, we have tested it on two different animations: "Head-Shaping" (Figure 47) and "Chicken Crossing" (Figure 46).

For each animation, we report the average number of bits per coordinate when using each one of the four predictors: space-only, time-only, ELP, and Replica. In order to demonstrate the dependency of these results on the lossy quantization, we provide results for four different quantizations of the vertex coordinates.

We illustrate the errors that result from these quantizations by showing several frames of the Chicken Crossing animation zoomed in at different degrees of quantization in Fig. 52.

Seven bits quantization breaks through the threshold of what is acceptable from a user point of view. Floating point precision is not necessary, for it is hard to tell the difference between the full precision and 13 bit quantization. For our results, we have decided to include 13 bit and lower quantization on our tests.

Table 3: Compression results in bits per coordinate for the Head Shaping animation. To avoid biasing the results by over-sampling in space or time, we use a sub-sampled version having 64 frames and 250 vertices.

Head Shaping	7 Bit	9 Bit	11 Bit	13 Bit
Space-only	3.07	4.94	6.98	9.16
Time-only	0.80	1.13	1.52	2.02
ELP	0.61	0.96	1.42	2.05
Replica	0.60	0.94	1.39	2.02

One may notice that the ELP and Replica predictors yield nearly identical results. Both are consistently better than space-only and time-only predictors.

Note that we have been using a zero-order time-only predictor because we limit our memory footprint to a single previous mesh. To be fair to time-only extrapolating prediction, we have compared below the zero-order time-only predictor to higher-order ones, computed as follows. Running the time-only predictor a second time on its residues raises it to a first order time-only predictor, which may also be computed as: $2*c.n.v.g(f-1)-c.n.v.g(f-2)$. Repeating this process a third time produces a second order time-only predictor, which may also be computed directly as $3*c.n.v.g(f-1)-3*c.n.v.g(f-2)+3*c.n.v.g(f-3)$. The results are shown in Table 5 for the Chicken Crossing data set. Notice that second-order time-only prediction does not improve upon first order. Nor do subsequent passes. First-order is significantly better than zero-order, but still not competitive with the space-time predictors.

To illustrate the dependency of the compression ratios on the time and space sampling frequencies, we have started with a very high resolution version of the Head Shaping animation with 6482 frames and $16,000$ vertices and have compared compression results for various combinations of sub-sampling in time and space. We have used the Replica predictor with 13-bit quantization. The results are shown in Table 6. Notice that, as expected,

Table 4: Compression results in bits per coordinate of the Chicken Crossing animation. Our proposed predictors show compression gains over previous approaches.

Chicken Crossing	7 Bit	9 Bit	11 Bit	13 Bit
Space-only	1.90	3.37	5.20	7.19
Time-only	1.78	3.29	5.03	6.91
ELP	1.37	1.79	2.28	3.01
Replica	1.37	1.83	2.35	2.91

compression results increase with sampling in both time and space. Furthermore, notice that our approach is capable of exploiting coherence in time when the animation is super-sampled in time but not in space; or coherence in space, when the mesh is over-sampled in space but not in time.

Table 5: Chicken Crossing animation compressed using only prediction through time, the results are in bits per coordinate. For an animation of this granularity, a prediction of second order does not achieve any improvement over first order.

Chicken Crossing	7 Bit	9 Bit	11 Bit	13 Bit
Zero-order Time-only	1.78	3.29	5.03	6.91
First-order	1.62	2.43	3.57	5.01
Second-order	2.19	2.96	3.91	5.07

Note that, as demonstrated by our experiment, although the benefits of time-coherence diminish with temporal sub-sampling, they are significant (58% savings when 3 frames out of every 4 are dropped, and 33% savings when 15 out of every 16 frames are dropped). Therefore, the proposed extrapolating predictors will be valuable, even if one were to use them to compress a sub-sampled set of key-frames and morph between the transmitted key-frames to restore the missing frames.

Table 6: Compression of the Head Shaping animation. Sampling over time and space affects the performance of prediction, increasing the compression cost.

Head Shaping	648 Frames	64 Frames	6 Frames
1/4 vertices	0.38	0.92	3.27
1/16 vertices	0.55	1.33	4.83
1/64 vertices	0.78	2.02	7.95

Comparing the compression results achieved with Dynapack to those of other previously published animation compression approaches has proven rather difficult, because the

reported results describe lossy compression and because the resulting errors, if at all reported, is measured using a variety of ways, which do not easily map into the quantization errors used by Dynapack, which guarantee a bound on the worst case Hausdorff error between the original models and the compressed ones. In spite of these difficulties, we can safely conclude the guaranteed maximum error of the animations compressed with Dynapack is comparable with the reported error in animations compressed with other approaches that would yield a similar compressed file size, although it is not clear that these reported errors describe the conservative worst case deviation (as we do) or a more forgiving least square statistical measure of it.

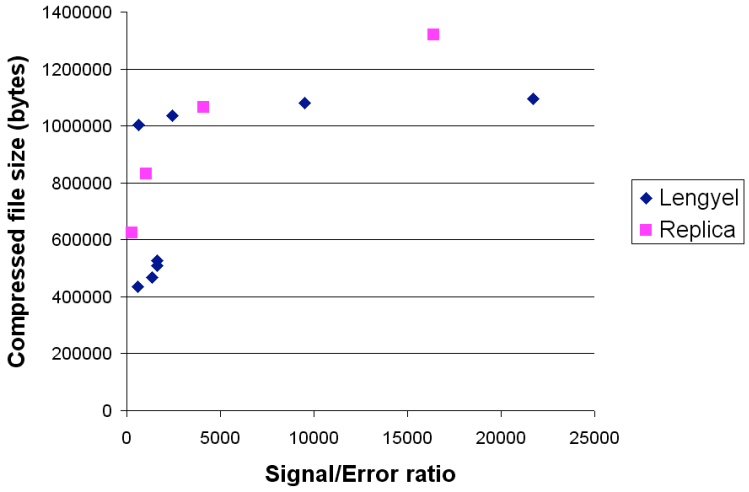


Figure 53: Comparision results with Lengyel's technique.

Fig. 53 shows a comparison of our method with the results obtained by Lengyel [64], for which the error was expressed in dB using signal/noise ratio. Note that one dB corresponds to $10 \log_{10}(Error/range)$. Our interpretation of the reported results is that Lengyel's approach yield 1.0 bit per coordinate when the model is highly quantized. Although Lengyel uses a different quantization from ours, the magnitude of the error he reports, which, to give him the benefits of the doubt, we assume to be the worst case error bound, is similar to the error we obtain when using an 8-bit quantization, for which Dynapack yields 1.5 bits per coordinate with Replica and 1.45 with ELP. Hence, for such over-quantized models Dynapack results in a 45% increase in storage over Lengyel's approach. Note that this penalty

may still be acceptable, and that one may chose to trade the better compression results of Lengyel for the simplicity of Dynapack. When using an 11 bit quantization, Dynapack compressed the entire animation to 1.06 Mbytes. Lengyel's approach produces a file of 1.03 Mbytes with a comparable accuracy. Hence, both approaches yield comparable results in this case. Lengyel does not report compression results that would match the accuracy of our 13-bit quantization. Also, Lengyel's result with more accuracy need 6.6 Mbytes, while a comparable compressed mesh of ours would result in 1.35 Mbytes (quantizing to 15 bits).

Alexa and Müller [2] do not provide an explicit error measure and the small size of the illustrations in their paper prevent us from estimating the accuracy of their compression. They report compression results between 3.85 bits per coordinate and 0.8 bits per coordinate. The 0.8 bits per coordinate animation shows errors that are significantly larger that those produced by Dynapack for 7 bit quantization.

We conclude that Dynapack with either the ELP or the Replica predictor yields results that are comparable to, and sometimes better than, results reported in recent animation compression schemes. Its strength stems from the simplicity of its implementation, which does not require preprocessing and works on streaming animations requiring only to buffer the previous frame.

The two animation sequences for which we were able to run the tests reported here do not demonstrate the benefits or Replica over ELP that we were anticipating. Still, we continue to believe that Replica has considerable advantages over ELP for animations in which large portions of the surface undergo major rotations and scaling.

6.6 Lossy Animation Compression

To increase the compression ratios obtained in Dynapack we explore lossy compression schemes. In cases where the initial models of the animations come from engineering models, or laser scans, the level of detail is far to large for regular uses. Animations with those characteristics can benefit from lossy compression.

Quantization applied to animated triangle meshes consist in quantizing the geometry of the animations. Quantization is a technique that introduces error globally. In the case of

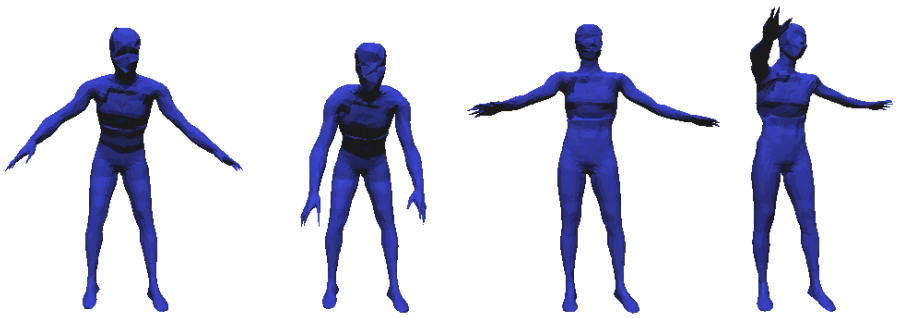


Figure 54: The figure depicts several frames of our **stretching** animation. These frames are the cutting points of the clips. Clippacker compressed this 1100-frame and 6983-vertex animation into 1.5 Mbytes, or 1.62 bits/vertex,frame.

triangle meshes, simplification is a better way to create a low cost approximation of a triangle mesh because simplification modifies the mesh of the animation adaptively, minimizing the error.

Each frame of the animation could be simplified [33] and compressed [88]. Simplifying and compressing each frame independently of the other ones produces unsatisfying results for two reasons. Firstly, the lack of coherence in the independent simplification of consecutive frames produces temporal discontinuities perceived as annoying, and often misleading, popping effects. Secondly, the temporal coherence of the animation is not exploited by the independent compression of frames. As we have seen in the previous section, using a previous frame to predict the next one may improve compression significantly. We propose to apply simplification to the whole of the animation, not a single frame.

6.7 Animation Simplification

We focus on mesh simplification that collapses edges of the triangle mesh one at a time (see Chapter 5). Since simplification (on its simplest form) only changes connectivity, it is straightforward to apply simplification methods to animations with constant connectivity. This process collapses an edge at a time in all frames of the animation; thus maintaining constant connectivity. However the selection of which edge to collapse next must take into account the errors introduced by the collapse in each frame.

The overall strategy of the simplification process follows the Garland's classic Quadric

Error Metrics [33]. Garland's method estimates the error resulting from the collapse [44] of each edge of G using quadrics. It then selects the edge with the lowest error and collapses it. The error of collapsing edge e is $E(e)$. We use $E_k(e)$, that represents the error of collapsing edge e in frame k. The process is repeated until the edge with lowest error exceeds a prescribed threshold.

Our method differs from Garland's in that the error of an edge collapse $E(e)$ is the maximum of $E_k(e)$ for all frames k. Also, our edge selection process is an iterative sweep. We traverse and collapse all edges with error lower than the threshold regardless of the posterior occurrence of an edge with smaller error.

As explained by Garland [33], each vertex of the mesh has a quadric associated. With the quadric it is possible to compute the sum of the square of the distances from a given point to a set of planes.

As explained in [34], for each normal n_i associated with vertex v, we compute the $3x3$ matrix A_i, the vector b_i, and the coefficient c_i. The coefficients of A, b, and c are computed as sums of the corresponding coefficients of A_i, b_i, c_i for all i. We save the result as the quadric error function of vertex v.

Given a vertex v and a normal n, A, b and c are computed as follows:

Consider a plane P through vertex v with normal n. Now consider a point u. the distance $d(u, P)$ between u and P is $|(u - v) \bullet n|$. The square of that distance (Q) may be written as $((u - v) \bullet n)^2$, which is $(u \bullet n)^2 - 2(v \bullet n)u \bullet n + (v \bullet n)^2$. It may be written as

$$Q(u) = u^T Au + 2Bu + C \tag{22}$$

where $A = nn^T$, $B = (v \bullet n)n^T$, and $C = (v \bullet n)^2$.

The quadratic error function of a cluster may be represented by its A, B and C terms. the representative vertex v^* of the cluster is computed by solving

$$Av^* = -B \tag{23}$$

When the system is degenerate, we compute v^* as explained in [67].

We propose to take a conservative approach to extend Garland's method to the simplification of animations to guarantee that all the frames in the animation have an error are lower than our prescribed threshold. We define the error of a collapse $E(e)$ that affects a series of frames $0..n$ to be the maximum error of the collapse of the edge in each frame:

$$E(e) = \max_{k=0..n} (E_k(e)) \tag{24}$$

This measure ensures that the total error of the simplification is never more than the threshold. One single frame where the error of an edge collapse exceeds the threshold and that edge cannot be collapsed.

To improve the speed of our clip simplification, we perform a single pass over all edges. For each edge, if its error is smaller than the threshold, it is collapsed. The order in which the edges are visited is random and we access each edge only once. A sequential traversal of the edges using a single pass results in non-pleasing meshes. Random order collapsing does not rule out the possibility of making wrong collapses, that possibility is just greatly reduced. Garland's method picks the edge with the smallest cost to collapse at each step.

We have implemented our clip simplification algorithm taking ideas from Streaming Meshes [51, 52]. The memory footprint our simplification needs is greatly reduced thanks to their streaming approach. Instead of loading all the meshes in memory, we can just load a small buffer per frame of the animation. Using a buffer of 5 to 20 percent of the size of the mesh, the memory usage is reduced to less than one tenth of its previous footprint.

The process of simplification of an animation is not optimal for large animations. Large animations might have edges with low collapse error in some frames, and large collapse error in other frames. Our method is conservative, and selects the error associated with each edge collapse as the maximum over all frames. In the case of large animations, the number of edges to be collapse under an error threshold is not as large as it would be if each frame were to be simplified independently. As discussed previously, independent frame simplification would add costs in the form of need to encode each connectivity, and possible visual artifacts.

We propose to segment the animation into *clips*, contiguous set of frames where the trade

off between encoding the added connectivities and the gains of simplification is optimal.

6.8 Animation Segmentation

The efficiency of Clippacker is directly related to the segmentation of each animation. If the clips produced are too large, the simplification process is constrained by some of the frames of the clip, and yields a model sub-optimally simplified. On the other hand, if the clips are too short, the overhead of encoding the connectivity and the large number of transition between clips overpasses the meager benefits of smaller clip simplification.

A simple approach such as creating clips of constant size would not take advantage of the smooth frames on the animation and would try to put together conflictive frames in sections of the animation with harsh transformations.

The search space for cutting a clip is exponential. Given a clip of length n, between each frame a cut could be introduced, so the all the cuts of the clip can be represented with a binary number of n bits, 1 if there is a cut, 0 if there is none. Thus, the search space is 2^{n-1}. Note that the optimal number of cuts in a clip is unknown.

The ideal way to obtain the optimal set of cuts is to try all possible combinations 2^{n-1} and pick the best. The running time of exhaustive search is exponential and it is not feasible. We propose to make the decision whether to cut in between two frames only once. Thus, we order the $n-1$ decisions. There are $n!$ possible ordering of the cuts. There are far more ordering of the cuts than possible combinations; once we pick an order we only need to take $n-1$ decisions. We propose two orders:

The **greedy** approach computes the optimal set of clips by deciding to merge the pair of clips with the largest decrement on cost. The decrement on cost of the merge of a pair of clips, A, B is computed as: $Cost(A) + Cost(B) - Cost(AB)$. We compute the decrement of cost for every pair of consecutive clips. We merge the pair with the largest decrement, and recompute the cost of merging with the neighbors of the new clip. For an animation with n frames and k merges, we compute the cost of a merge $n - 1 + 2k$ times.

The **incremental** approach segments an animation in a similar way. The decrement of cost of merging a pair of clips is computed the same was as in the **greedy** approach. The

incremental approach tries merging the first pair of clips, from left to right. If a pair of clips has a positive decrement of cost, they are merged. Otherwise, the left-most clip is output as a final clip in the segmentation, and the algorithm does not consider it anymore for merging. For an animation with n frames, the **incremental** approach computes the cost of merging $n-1$ times.

6.9 Animation Transition

Two consecutive clips have different connectivity; otherwise they would have been merged. The transition between the two clips, due to the change in connectivity and thus in geometry, can produce popping effects. We have investigated two approaches to mitigate the possible popping artifacts.

The first approach consists in performing a geomorph between the connectivity of the two clips. Geomorphs require a correspondence between the two meshes. We have decided to encode the simplified connectivity for each clip, thus we do not use geomorphs.

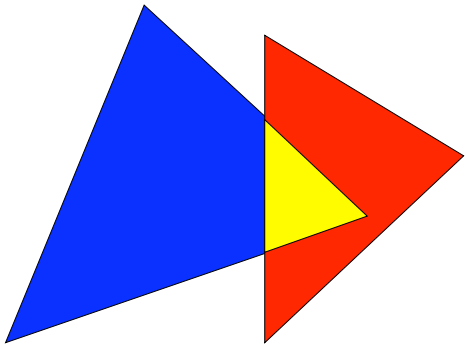


Figure 55: Alpha blending clip transition scheme: blue as clip1, red as clip2. For a given α, the blue region is computed as $\alpha * clip1_{color} + (1-\alpha) * background$. The red region is computed as $(1-\alpha) * clip2_{color} + \alpha * background$. The intersection of both clips, the yellow region is computed as: $\alpha * clip1_{color} + (1-\alpha) * clip2_{color}$.

The second approach is alpha blending. Alpha blending is a well known technique to blend a polygon with a background. When there is an overlap of transparent polygons the order in which they are rendered defines the output. That behavior is changes so the morphing remains the same no matter what clip is closer to the screen. Alpha morphing is the process to superpose the two clips using alpha blending and progressively fade out the first clip as the next clip fades in. Where two clips overlap the result is a blend of both

clips, without taking the background into account. Where only one clip is rendered, normal alpha blending is performed.

Pixels where only polygons from clip1 are rendered are computed as $\alpha * clip1_{color} + (1 - \alpha) * background_{color}$. For those pixels where the two clips overlap, the final color is computed as $\alpha * clip1_{color} + (1 - \alpha) * clip2_{color}$. Finally, pixels where only polygons from clip2 are rendered are computed as $(1-\alpha)*clip2_{color}+(\alpha)*background_{color}$ (see Figure 55).

For animations with rough sampling over time, the lack of continuity between consecutive frames causes already some popping artifacts. In such case it is not recommended to apply any transition, the already existing artifacts do not disappear, and alpha morphing can introduce artifacts in the form of ghosts.

6.10 Lossy Animation Compression Results

We have applied Clippacker to three datasets. Our first two datasets are motion capture animations. The motion capture data is used to deform a human skin mesh. Please note that the motion capture was not perfect, and the animations have some irregularities. The Stretching animation represents a set of morning stretching exercises, the animation has 1100 frames. The animation has been simplied up to a 0.1 RMS error, computed with metro [21]. The PickBox animation represents an actor walking and picking up a box, the animation has 170 frames. The simplification error threshold for this animation is 0.15 RMS. Both animations use a human skin of 6983 vertices. Motion capture animations represent the target kind of animations Clippacker has been designed to compress.

Table 7: Table with results and compression rates for Clippacker. The greedy approach outperforms the incremental approach.

Dataset	vertices	frames	Greedy	bits/v,f	Incremental	bits/v,f
Stretching	6983	1100	1,564,735	1.62	1,582,667	1.64
PickBox	6983	170	124,553	0.83	176,205	1.18
Horn	10242	260	169,700	0.5	177,136	0.5

Our third dataset is a synthetic animation. It's the morph of a ball into a ball with horns. The animation has 260 frames and the mesh has 10492 vertices. It has been simplied up to 0.005 RMS error. The results on this synthetic animation show the potential of our

approach, yet these results are not representative.

Other animation papers use well known sequences like the chicken, or the face for reference. Our method makes intensive use of simplification. The latter animated meshes are simplified alreeady, they are not in the kind of animations Clippacker is suited to compress.

The horn dataset is a good example of a mesh with deforming parts that a straight forward compression is not enough, our partition technique divides the animation into 9 different clips.

Comparing animation compression is not an easy task because each paper uses its own dataset, and its own error measure. We have measured our error with metro, as Karni and Gotsman [57] have done. Their paper is amonst the most recent in animation compression. Our technique compares to them, they achieve 1.0 bits per vertex and frame with an error of 0.08 RMS. With 0.1 RMS we obtain 1.6 bits per vertex. Clippacker obtains results similar, yet worse than their technique. On the other hand, clippacker scales well to larger animations, and larger meshes.

6.11 Encoding a simplification

Simplification, as described by Hoppe [44,45] encodes the connectivity of a progressive mesh in 10.4 bits per vertex. The position of the vertices in the simplified version of a mesh is defined by optimization (error minimization) and is independent of the order of collapses. We have explored the possibility of compactly encoding the simplification process.

The benefits of knowing the simplification process for a clip is that it provides a mapping between the geometry of the clip and and the geometry of the original connectivity. Through the original connectivity, it is possible then to associate the different geometry of two clips. Such relationship is useful to improve prediction, to compute smooth geomorphs, and to transmit mesh properties such as texture coordinates.

Let M be a mesh, and S its simplification through a series of edge collapses. Pain all edges of M blue. Each edge collapse step of a simplification process collapses an edge of M, merging the two vertices of the edge. Paint in red all edges of M that have been the subject of an edge-collapse operation in the simplification process that produced S. Each

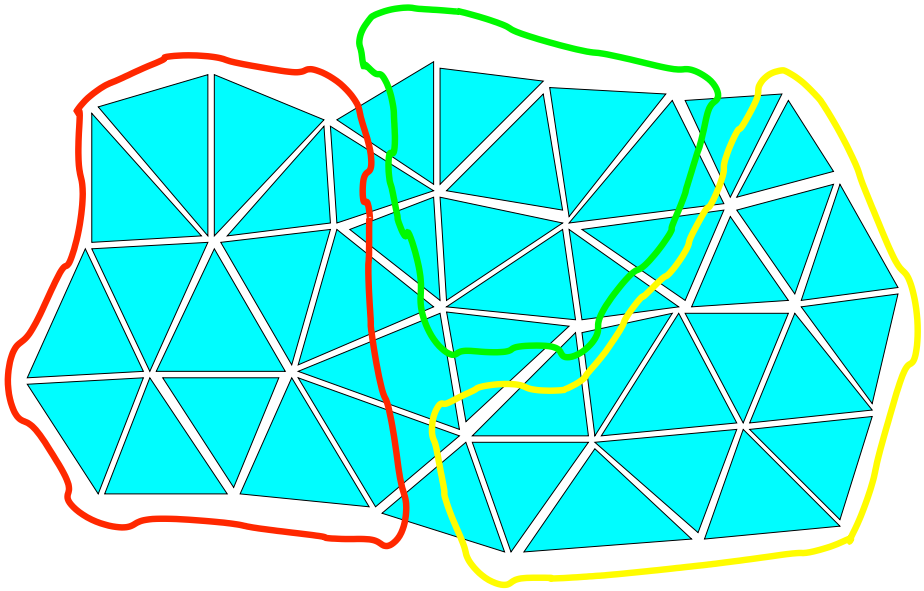


Figure 56: Triangle mesh that has been simplified to one single triangle, three vertex clusters marked red, green and blue.

maximally connected component of the set of red edges of M defines a cluster of vertices of M. All the vertices of a given cluster c are collapsed into a single representative vertex v_c of S. Each vertex of S is representative of a different cluster of M, even though some clusters may contain a single vertex.

Paint in green the non red edges that connect two vertices of the same cluster. All edges that are neither red or green remain blue. Paint blue each triangle bounded by 3 blue edges. Note that collapsing all the vertices of each cluster c of M into the corresponding representative vertex v_c, will stretch each blue triangle of M into a triangle of S and will collapse all other triangles.

Note that the connectivity of S is completely defined by the clustering of the vertices of M. Furthermore, the clustering is completely defined by the set of red edges. Hence, the red edges of M define the connectivity of S.

Furthermore, note that the clusters, and hence the connectivity of S, are independent of the order in which the edges of M are painted red, and hence of the order in which the edge-collapse operations are executed; assuming that this order does not affect the final position of the representative vertex.

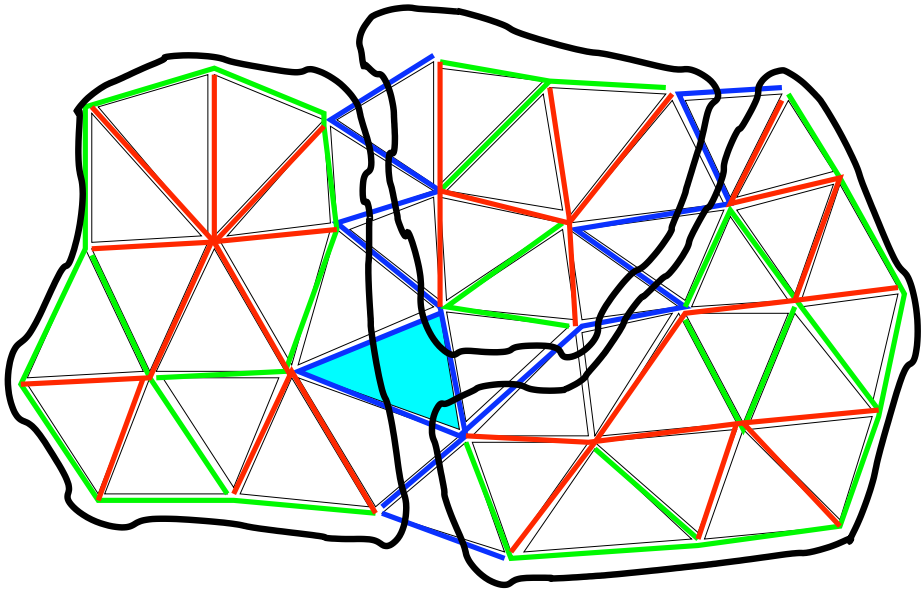


Figure 57: For each cluster of vertices, red are the collapsed edges, green the edges connecting vertices in the same cluster and blue the edges connecting edges of different clusters.

Finally, note that a connected set of red edges is always a spanning tree of the vertices of the cluster. Consequently, a cluster with more than 3 vertices may usually be defined in more than one way by a spanning tree of red edges, and the same cluster could have been defined by any vertex spanning tree of red or green edges.

In summary, we have stablished that:

1. Each vertex of the simplified mesh S represents a cluster of the vertices of the original mesh M.

2. The clusters of S are defined as the maximally connected sets of red edges and are hence independent of the order in which the edges of M have been collapsed.

3. Each cluster may be represented by any spanning tree of red or green edges that connects its vertices.

All combinations of such spanning trees for all clusters and all permutations of the order in which the spanning tree edges are collapsed define a family of simplification processes that produce the same connectivity for S.

Theorem 2. *Given an original mesh M and the list of clusters of vertices of M the simplified mesh S is uniquely defined.*

Proof. Two meshes are equivalent if their geometry and connectivity is equivalent. We divide this proof into geometry and connectivity:

- **Geometry.** The number of vertices in S is the number of clusters in the simplification. The number of clusters is independent of the way clusters are defined. The position of each vertex is defined by the function that minimizes the total error, the sum of the square of the distance from the vertex to each plane. The planes used in the minimization error are the planes of the triangles containing a vertex from M in the error minimized vertex's cluster. The order in which the triangles were added to the cluster do not influence the set of planes; the final position of the representative of the cluster is independent of the way the cluster is defined. Thus, the geometry of S is equivalent constant despite of the way clusters are defined, despite the order in which the edges are collapsed.

- **Connectivity.** The connectivity of a mesh is defined by the vertices of the mesh and the edges between the vertices. All the versions of S have the same vertices (number and position). There will be an edge between two vertices v_1 and v_2 of S if and only if there was an edge between a vertex of v_1's cluster and a vertex of v_2's cluster in M. Thus the existence of an edge between two cluster representatives in S depends on the existence of an edge between a pair of vertices of the clusters. The edges of S are independent of the way clusters are defined.

□

6.11.1 Encoding

We define a canonical way to encode the vertex clusters of a simplification. The canonical encoding of a simplification consists in selecting for each cluster, a set of edges that connects all vertices in the tree. The graph defined by the selected edges cannot contain cycles. The

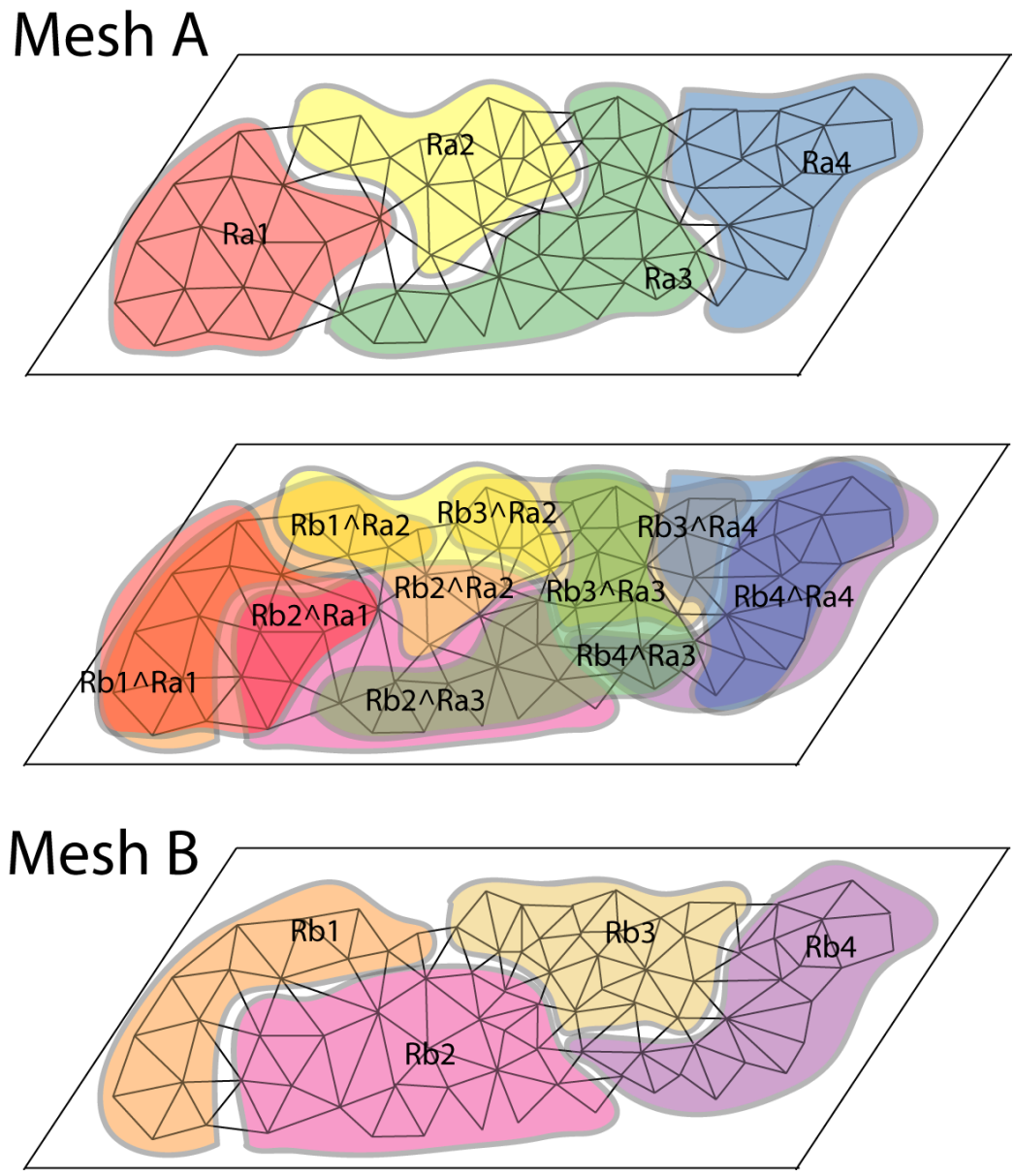


Figure 58: Mesh A is simplified by the shown clusters, as well as mesh B. To produce a geomorph, we use the simplification information to create an intermediate mesh, shown in the middle, where each vertex of the intermediate mesh belongs to the intersection of clusters from mesh A and mesh B.

edges are chosen in the lexicographical order defined by the original connectivity. Therefore, any two equivalent simplifications have the same canonical encoding, the same set of edges defining each cluster.

To encode the simplification, we encode a symbol per each triangle. The symbol *T0* marks a triangle to be left intact. The symbols *C1, C2, C3* define that a triangle will have a collapse of its first, second or third edge respectively. Let us assume a simplification that reduces the number of triangles T to one hundredth of T. Only one percent of T triangles will encode the symbol T0, the rest will be encoded using C1, C2 or C3. For each Cx symbol encoded, two triangles will be collapsed, the marked one and the triangle sharing the collapsed edge. Hence, we the number of Cx symbols is half the number of collapsed triangles. The entropy of the symbols when the simplification collapses 99% of the triangles is 1.69 bits per symbol, that becomes 0.89 bits per triangle because we encode close to half as many symbols as triangles. Our encoding reports the maximum bits per triangle encoding of 1.2 when the mesh simplifies 70% of its triangles.

Hoppe's approach to encode the simplification information [45] encodes a very simplified mesh, and a sequence of operations that refine the mesh to its original state. We propose to encode the original connectivity, and we encode information to simplify it. Hoppe's cost is much higher than ours, by a factor of ten; on the other hand, with the original connectivity the simplification information we encode is redundant. Our method is useful when computing the simplification with its error minimization is not possible or cost effective, and when it is desired to create a map between two or more different simplifications. We review some of the applications that benefit from our technique in the next subsection.

6.11.2 Applications

Simplification encoding can be used in combination with Clippacker. Clippacker produces several clips with different connectivity. All the clips are simplifications from a common connectivity, instead of encoding each simplified connectivity on its own we can encode the simplification process from the original connectivity for each clip.

Encoding the simplification allows us to create geomorph between two clips. Each clip

has a different simplification, thus a different set of clusters. Figure 58 depicts an example. Two different simplified connectivities are shown, mesh A and mesh B. We can compute the intersection of the set of clusters because we encoded the simplification information. The computed connectivity, shown in the middle of the figure, has the following property: each vertex belongs to exactly one cluster in mesh A and exactly one cluster in mesh B. The connectivity of the intermediate mesh is a plausible pre-simplficiation for both mesh A and mesh B; a common refined version of both meshes. The transition between mesh A and mesh B is achieved by replacing mesh A with the intermediate connectivity, in such a way that each vertex of the intermediate connectivity is in the location of the vertex of A it corresponds. Each vertex of the intermediate connectivity linearly translates from the start position to the position of its correspondent vertex of mesh B. After the intermediate mesh coincides with mesh B, mesh B is used onwards.

View dependent simplification is a process that simplifies based on the error to a view, instead of overall error. View dependent simplification is used to explore very large meshes that a client application cannot show in real time; the process removes detail that is not shown on screen. Simplification encoding can be used with view dependent simplification. A set of simplifications is encoded for different view positions. The client has a current simplification, and receives a new simplification when the viewpoint moves from an area. Two simplification encodings that correspond to adjacent areas are similar. Each simplification encoding is based on our canonical order, to transmit a new encoding we transmit the XOR of the list of symbols, taking advantage of the similarity between encodings.

6.12 Conclusion

Dynapack is a compression scheme for the 3D animations of triangle meshes of constant connectivity that undergo arbitrary deformations. Because Dynapack requires only accessing the previous frame when compressing or decompressing an animation frame, it is particularly well suited to real-time compression, out-of-core compression, and decompression of streaming animations. Due to its traversal, and prediction based on addition and subtraction, it may prove to be a good candidate for a hardware assisted decompression. Dynapack

may be implemented using a trivial algorithm that traverses the triangles of the mesh. It supports two space-time predictors. The Extended Lorenzo predictor (ELP) reduces to nearly zero the cost of encoding portions of the animations where a subset of the mesh undergoes a pure translation. Its main advantage lies in the fact that the predictor formula uses only point additions and subtractions. The more elaborate Replica predictor extends the nearly zero-cost prediction capability to combinations of all rigid body motions and uniform scaling transformations. The performance of both decays gradually as the behavior of the mesh departs from these simple transformations and as the space and time sampling density is decreased. Still, even for subsampled meshes, these predictors are superior to space-only and to time-only predictors, and hence will benefit other compression techniques that sub-sample the data and rely on interpolation for restoring the missing frames.

Clippacker is a lossy animated mesh compression that introduces error in a controlled way through simplification, and segments an animation to maximize the compression ratio. Clippacker has been designed to be effective in long sequences of animation where several different deformations occur along time. It automatically finds the best place to segment the animation and provides a smooth transition. Long sequence of human motions fall into this category.

We have proposed a scheme to encode the simplification information, developing theory stating the order independence of a sequence of edge collapses. We achieve a compact encoding of no more than 1.2 bits per triangle of the unsimplified mesh. Simplification encoding allows for the mapping between vertices and faces of different meshes, its scope far outreaches the compression focus of Clippacker.

Simplification encoding is a novel contribution, proving properties about simplification and achieving a compact representation of no more than 1.2 bits per triangle. Simplification encoding can be used for geomorphs and view dependent simplification encoding and transmission, amongst others.

In the case of animation mesh compression, it is common to perform a simplification leaving only 5% of the original triangles. Such drastic simplifications render the cost of encoding the smaller connectivity cheaper than the cost of encoding the simplification. In

our Clippacker test datasets we decided not to apply simplification encoding.

CHAPTER VII

ISOSURFACE SURVEY

An isoset, as explained in Chapter 2, is the set of points where a function takes a given value v:

$$S_v = \{p | f(p) = v\}$$

Figure 60 shows examples of common isosets. A 2D weather map shows isosets where the pressure is constant.

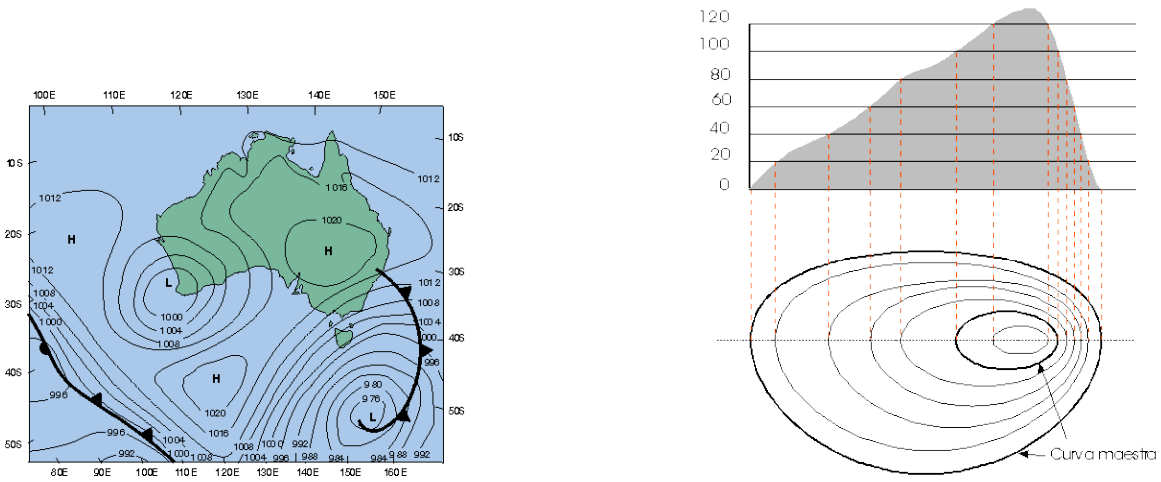


Figure 59: Examples of isosets. On the left it is depicted a weather map from Australia, courtesy of the Bureau of Meteorology of Commonwealth of Australia. On the right there is an elevation map from a terrain, courtesy of the Free Encyclopedia, spanish version.

Isosurfaces in 3D are used in medicine, for example to visualize a patient's MRI scans. Furthermore, engineering models and physical simulations are often visualized using isosurfaces which may for example represent the points where temperature is equal to isovalue v.

The isosets are extracted from a sampled version of the function f. f completely defines the isosets, but the extraction of isosets from a sample of f is limited to a sampled version of the isosets. The surface that the isosets define is not properly defined in all cases.

Space is divided into cells and we assume a simple model of f in each cell (bilinear in 2D, trilinear in 3D). An isosurface is a triangle mesh approximating the derived isoset in the cell. Some schemes position the vertices of the triangle mesh on edges of the cell. Others (Dual contouring [56]) have at most one vertex in each cell. The former triangle mesh must be manifold, although its topology is not always uniquely defined [80].

As seen in Chapter 2, a **node** is a point of the regular grid with a scalar value. We refer to a node being **in** when its value is larger than the isovalue, and **out** if it is smaller. A **cell** is a cube having 2^n neighboring nodes as vertices. Four nodes make a cell (square) in 2D, eight nodes make a cell (cube) in 3D. A **link** is a line segment between adjacent nodes. A link connecting two nodes, one *in* and the other *out*, is called a **stick**. The point where an isosurface intersects a stick is a **vertex**.

Extracting a triangulation of the isosurface from the regular grid that samples the implicit function is a computationally expensive task. It was first approached by Lorensen et al in *Marching cubes* paper [71]. Marching cubes process the regular grid in a scanline order. For each cell that is intersected by the isosurface, marching cubes generates the triangles of the isosurface contained in the cell. A cell in 3D has eight nodes, and each node can be either *in* or *out*, therefore there are 256 different possible cell configurations. Marching cubes computed the triangles for each cell configuration and stored them in a look up table.

Marching cubes has been extended in several ways [78, 79]. A very small fraction of all the cells in a grid are intersected by the isosurface (typically $O(n^{(d-1)/d})$, where n is the number of cells and d the dimension), yet marching cubes visits all cells. Approaches that deal with isosurface seeds [10, 17, 75, 102] precompute a small subset of cells to speed up the extraction process. To compute the small set of seeds a reeb graph or contour tree is used [17, 75]. The reeb graph stores the critical points of the data, where connected components of isosurfaces merge or are created.

Dual contouring [56] is a method for extracting an isosurface retrieving Hermite data (i.e; exact intersection of points and normals). Using this technique the isosurfaces that are extracted from a regular grid maintain their original features, and allowing sharp edges.

The vertices of the isosurface in this technique are not positioned along a stick, but are inside the cell. The position of the vertex is computed minimizing quadratic error based on normals.

One single isosurface is not enough to fully describe the implicit funciton, it captures a slice of the sampled function's behavior, the slice correspondent to the isovalue v. Typically, several isosurfaces have to be extracted from the regular grid. Extracting an isosurface is a costly process, individual isosurface extraction using marching cubes cannot be done in real time with moderately large datasets. We define (as seen in Chapter 2) an IsoSurfaceRange to be the set of isosurfaces whose isovalue is in a given interval:

$$\{S_v | v \in [v_1, v_2]\}$$

Bajaj's et al [10] build a set of seeds, subset of all the cells from the regular grid. For any connected component of any isosurface in the dataset, there is one cell in the set of seeds that belongs to the connected component. The set of seeds is built by adding all cells to the set of seeds, and later removing for each isosurface loop, all but one cell, making sure that no connected component is left without representation.

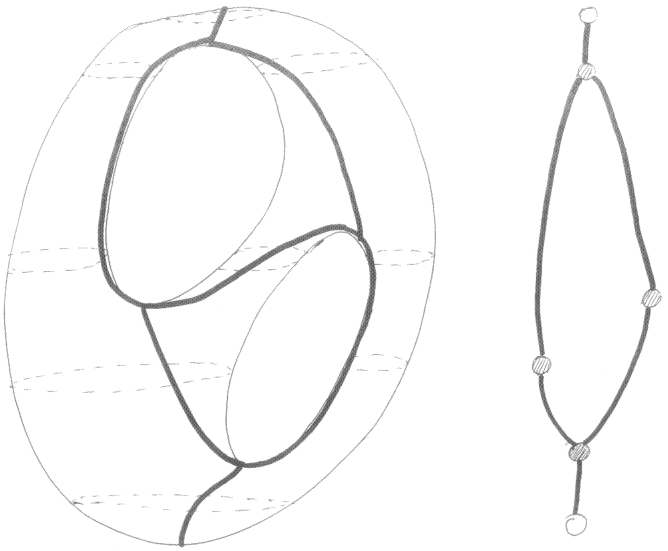


Figure 60: Reeb graph of a bitorus and some cross sections. Each contour determines an equivalence class which is represented in the Reeb graph by a single point. This structure is used to facilitate isosurface extraction.

Kreveld et al [102] proposes an approach that computes the contour tree of the mesh, and uses the tree to compute a minimum size seed in polynomial time and storage. The computation time used for the contour tree creation is quadratic.

Recent work by Carr [17] extend the use of contour trees to build a map of the shortest path between isosurfaces. His work allowed for models of moderate size (512^3) to be explored in real time.

Taubin [98] proposed an isosurface compression approach inspired in the JBIG image compression standard. He assigned to each node of the grid a single bit, indicating if the node was *in* or *out*. This bitmap was transmitted using a context encoding of 7 bits, corresponding to the in/out status of the 7 neighboring nodes in 3D. This information is enough to single out the sticks. His approach continues with the transmission of the refined position for each vertex along its stick, in a scanline order, also using a context encoder. For the stanford bunny dataset, Taubin reports 0.5 bits per face. That encoding does not specify the position of each vertex on a stick, the middle point is used. Taubin reports that each bit used to refine the position of the vertices on the sticks increase the bits per face by one bit.

Lee et al [63] proposed an approach for the compression and progressive transmission of isosurfaces. Their method computes an octree that holds the vertices of the isosurface. The octree is transmitted progressively, containing information of the topology and coarse geometry approximation of the isosurface. When the octree has been transmitted, the remaining geometry information is sent as normal and tangent corrections. Eckenstein et al [27] extended Lee's approach to compress time-varying isosurfaces. Their approach follows the footsteps of MPEG compression [55]. They define the isosurface as Lee did, as an octree. For each new isosurface, the algorithm encodes how to obtain the new octree from the old one. They divide the octree into blocks of 8^3, and encode displacement vectors. Lee et al report a compression cost of 6 to 7 bits per vertex. The encoding is not lossless, the geometry is transmitted as quantized corrections in the normal and tangent planes.

The previous techniques encode the position of the sticks where the isoset vertices lay, and approximate the position of the vertices. The values at the nodes are often stored

in floating point precision, the position of the vertex in a stick thus requires also floating point precision to be completely lossless. It can be argued that the precision of floating point numbers is not required for the error introduced in the position of a vertex in the stick, yet for scientific purposes, completely lossless compression is required. In cases where approximations are accepted, Taubin [98] proposed the use of four bits for each stick, arguing that it is seldom necessary more precision for visualization purposes.

CHAPTER VIII

ISOSURFACERANGE COMPRESSION

We propose an scheme for the on-request transmission of isosets. For example, in a 2D case, the user is exploring a map of Georgia. He requests an isoset, all the points at elevation h. He wishes to explore more, to find interesting hiking places. He keeps requesting isosets, some close to the previously transmitted, others further away. We design a system that transmits isoset data on demand, and takes advantage of the data already on the client's side.

In Chapter 6 we discussed approaches to compress sequences of meshes. Although we represent isosets as triangle meshes (sometimes as curves), we cannot use the techniques from Chapter 6. Our animated mesh compression techniques were based in meshes with temporal coherence (each mesh followed the previous one in time), and all the meshes shared the same connectivity. The approach we propose here is independent of the connectivity, and can transmit isosets in any order.

We transmit the minimal subset of values that completely determine the isoset requested by the client. Our approach is based on the traversal, identification and transmission of nodes and their associated value. Although the client might extract edges or triangles from the nodes to visualize the isoset, this task has no effect on transmission cost.

The following sections explain our approach in the 2D case. We cover how we generate and encode an isoset from a regular grid in Section 8.1, and we show results of our approach in Section 8.2. There are few differences between the 2D and the 3D approach, those will be describe in Section 8.3.

8.1 Isoset generation

Our isosurface generation (extraction) method is very similar to that of marching cubes. As previously explained, doing an exhaustive search of the regular grid is not convenient. We assume that a pre-computed a set of seeds so we only access the cells intersected by the

isosurface.

The encoder and decoder perform the same traversal hence we discuss them together. Our approach traverses each component of the isocurve. It starts with a seed stick and encodes/decodes its two node values (A, B from Figure 61).

The encoder and the decoder process need to store in memory the values at the nodes that determine the isosurface. Our implementation uses a quadtree in 2D, and an octree in 3D to store the nodes and their values. We generalize the quadtree and octree to our *Space Tree* structure. This structure provides fast, adaptive storage, while maintaining spatial relationships between the nodes.

Each cell is processed equally, from a stick and two known values (see Figure 61), our approach traces the isocurve and determines which link is the stick the curve exits through. While the isocurve never bifurcates, it is possible for a cell to have four sticks, the isocurve traversing the cell twice. In such case, we use a heuristic to select the behavior of the isocurve inside the cell.

The two nodes at the entrance stick are known to both the encoder and the decoder, but the other two nodes of the cell might not be known (see Figure 61).

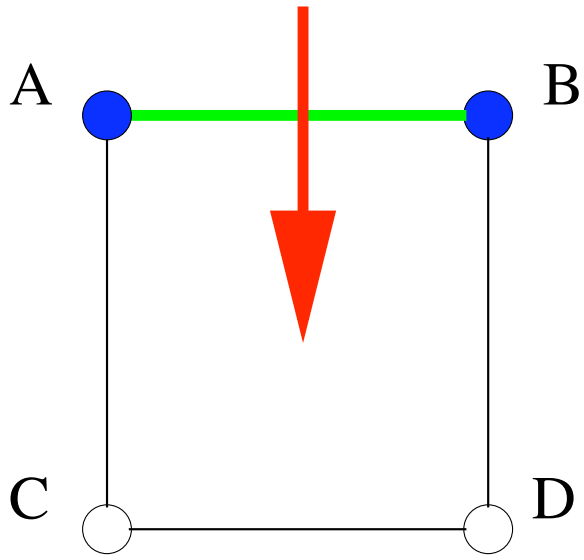


Figure 61: Example of a cell in 2D. Green is the entrance stick, blue represent two already encoded nodes, and red is the traversed part of the isosurface.

There are three possible cases when we process a cell (the rest are rotation and mirroring

of these cases):

1. *Three nodes from the cell are known*, the two nodes from the entrance stick and another. Without loss of generality, we assume that C is the other known point. If $\overline{AC}$ is a stick, then $\overline{AC}$ becomes the exit stick. If it is not a stick, we encode/decode D, and test to see which edge is a stick, $\overline{BD}$ or $\overline{CD}$. The edge that is a stick will become the exit stick.

2. *Only the two nodes from the entrance stick are known.* We use our prediction to guess the values at the remaining nodes: C' is the prediction for C and D' is the prediction for D. If $\overline{AC'}$ is a stick, we encode/decode the node C, and this becomes case 1. If it is not a stick, we check $\overline{BD'}$ for stick. If it is, we encode/decode D and go to case 1. If neither $\overline{AC'}$ nor $\overline{BD'}$ is a stick, we compute the vertex from the entrance stick; the point where the isosurface intersects the entrance stick. We encode/decode then the point closest to the vertex. If the vertex was closer to B than to A, we would encode/decode the node D. We proceed with case 1.

3. *All nodes in the cell are known.* If there is one single other stick, that becomes the exit stick (nothing needs to be encoded). There are cases when every edge is a stick. We call that kind of cell an *x-cell.* X-cells are ambiguous, there are two possible correct solutions to the trace of the isosurface in an x-cell (see Figure 62). We proceed in a similar way as case 2. We compute the vertex at the entrance stick. If the vertex is closer to B than to A, $\overline{BD}$ becomes the exit stick.

We disambiguate using a heuristic, we use the node closer to the vertex of the entrance stick to determine what stick to predict or to trace the curve to. Other heuristics that might be used here need trace of the isocurve previous to the entrance of the current cell, and extrapolate the path of the curve.

To encode the value of a node, we use our previous method from Chapter 4, Spectral prediction. The structure where the nodes and values are stored, the quadtree, provides a mask specifying which values are known. We consider a 5×5 neighborhood around the

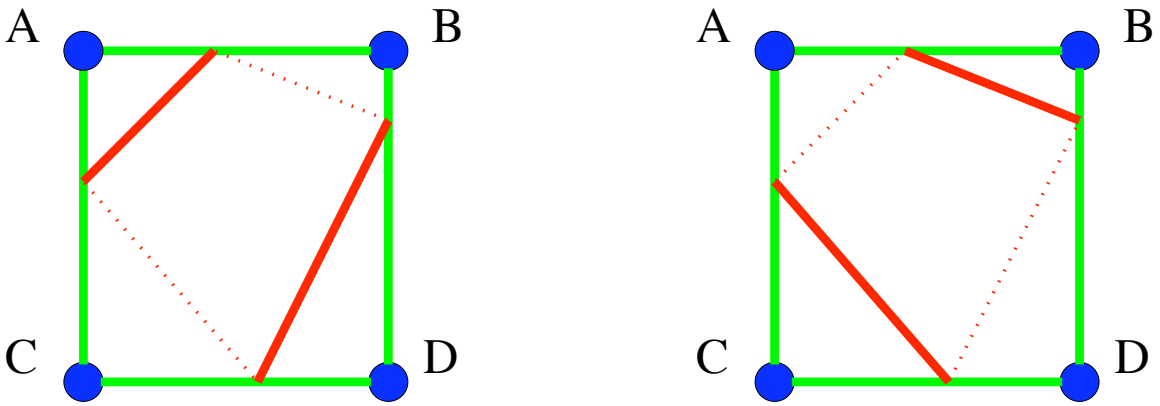


Figure 62: Ambiguous case in 2D, *x-cells*. Both isosurfaces are correct.

predicted node. The neighborhood for our Spectral prediction is 3×3; we can apply nine different spectral predictions. The 3×3 neighborhood with more known values is chosen to be the applied spectral prediction (as discussed in Section 4.4.5.2).

Using the real value and the prediction, we encode the residual as described by Isenburg and Lindstrom [68].

The position of the seeds is not known to the decoder, the seeds are transmitted in a different stream in raw format. For large isosurfaces, the ratio of seeds to encoded values is small. A seed is encoded as a link of the regular grid that intersects the isosurface. A seed is encoded as the position of a node of the link and symbol describing in which direction the second node of the link lays. In 2D, the discriminating symbol needs only one bit. The start of the decode process with a seed needs to bootstrap the traversal process by decoding the two nodes incident to the seed stick. In the case that there is no nearby point to predict the first node, we use the isovalue; the node's value is bound to be close to the isovalue. The second node already uses Spectral prediction. In the case that only the first node is known with value w, we linearly extrapolate the second node's value using the isovalue v; the prediction is $2v - w$.

It is important to point out the fact that our approach never sends the value of the same node twice. If a previous isocurve encoding had transmitted a node value, all following isocurves that need such node will have it available locally and will not need to encode it again.

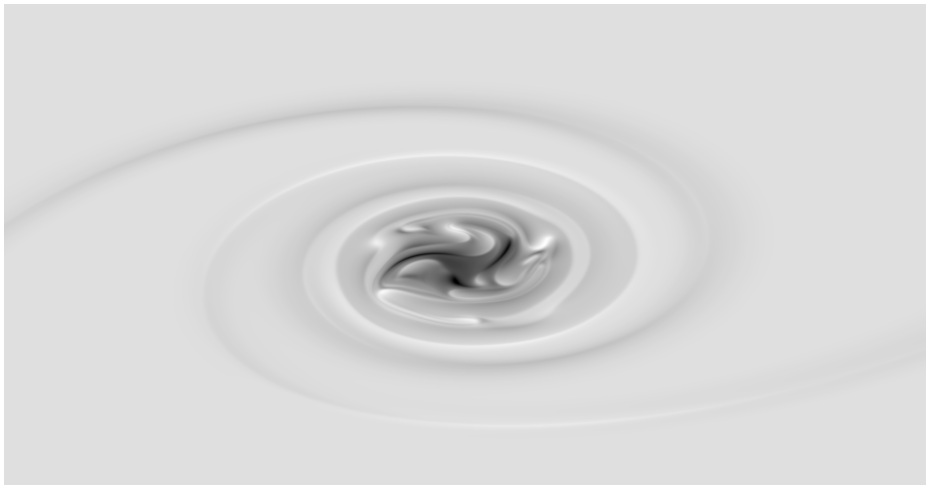


Figure 63: Vorticity dataset. 2D regular grid of 1025×5000, from [23]. This is a snapshot of a fluid simulation.

8.2 Results

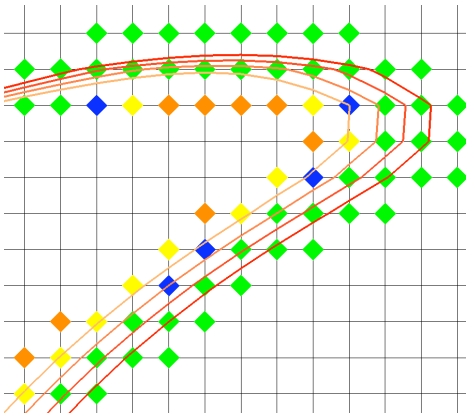


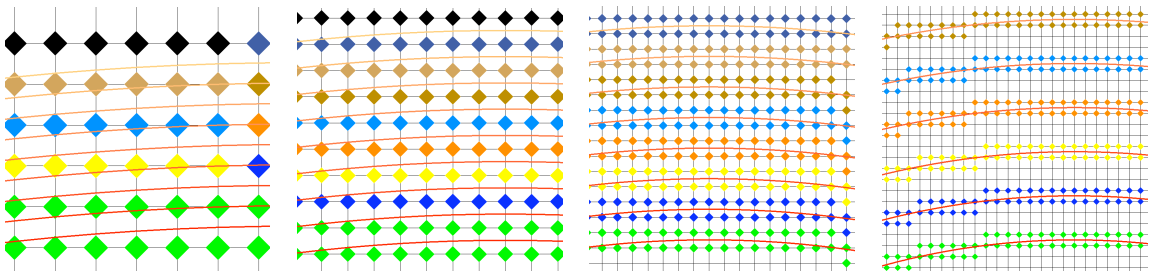
Figure 64: Zoom in a set of decompressed isosurfaces. The nodes transmitted with each isosurface are colored different.

We have tested our approach on two datasets. The first dataset is a distance map, its isocontours take the shape of a circle. A few renderings of that dataset are in Figure 65. We use the scientific dataset vorticity from Cook et al [23]. It has 1025×5000 size. It is the product of a fluid simulation. We have rendered it using white for the highest value and black for the lowest, see Figure 63. We use the circle dataset as a proof of concept of our ideas, and we use the vorticity dataset to show the effectiveness of our technique with real world data.

We transmit the nodes defining an isocurve, but it is common to refer to the compression

of isocurves by the number of vertices the isocurve has. From here on, we refer to *bits per node* to the average number of bits used to encode the node values transmitted, and *bits per vertex* to refer to the average number of bits per vertex of the transmitted isocontour.

Our technique can be used to compress one single isosurface. The performance achieved in that case should not be compared to regular isosurface compression because we encode more information than a single isosurface. For the circle dataset, our technique achieves 17.13 bits per vertex. Considering that the circle is a smooth dataset, that value is rather high. Our method achieves 12.1 bits per node, that is considered normal for a smooth dataset. In the 2D case, while compressing a single isosurface we use an average of 4.2 known neighbors. Due to our 3×3 neighborhood and the fact that the curve of the circle goes forward, we apply most of the time a prediction that is linear.



(a) small *(b) medium* *(c) large* *(d) very large*

Figure 65: From our synthetic circle dataset, four set of isosurfaces. From left to right, the step between isovalues is doubled. Each isocontour is colored with a different color, the points drawn are transmitted with each isocontour. It can be seen how closer isocontours transmit less points.

Our method's strength shows when we compress several isocontour. Figure 65 shows four set of isocontour. As can be seen in Figure 66, when compressing a set of isocontour that are very far apart from each other, the total result is just the compression of several single isolated isocontours. As the distance between the different isocontours draw closer, as seen in the *large*, *medium* and *small* set of isocontours, the cost of transmitting each new isosurface is reduced. The number of average neighbors used in prediction was 6.5, 7.3 and 7.4 for the datasets from *large* to *small*. The ratio between nodes and vertices was 0.5, 0.7 and 1.2 respectively. The *large* dataset does have almost the same node/vertex ratio as the individual isocontour, but its prediction is improved by having nearby values

from previously transmitted isocontours. As long as the new isocontour is nearby already decompressed data, our technique offers improvement. The total compression bits per vertex for the *very large*, *large*, *medium* and *small* is 17.28, 13.64, 6.49 and 4.13 respectively.

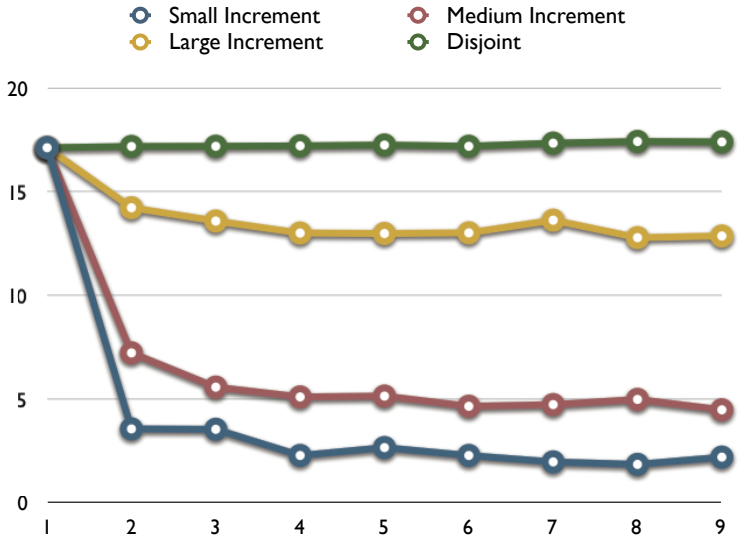


Figure 66: Bits per vertex for each set of isosurfaces from Figure 65. The x axis is the number of the transmitted isosurface. This shows the improvements of transmitting isosurfaces that overlap with previously decoded data.

As a result of the way the nodes are predicted, the order in which isocontours are encoded affects the compression ratio. For sets of disjoint isocontours the order has no influence on compression, but for sets of close isocontours it does affect the total compression ratio. For the case of the *medium* set of isocontours, we have ordered them to illustrate an extreme worse case scenario: the first half of the isocontours are transmitted in an order where there is no overlap, even though there are enough nearby points to help prediction, and the second half are transmitted but few nodes need to be send. The first half have results similar to the *large* set of isocontours, the second half has near perfect prediction (an average of 8.0 used neighbors), 0.8 bits per vertex. The total compression cost for the incremental *medium* set of isocontours is 6.49 bits per vertex, the cost of our order is 8.22. Order matters when compressing a dataset, on the other hand, the order is defined by the user.

For the vorticity dataset, from on a scientific simulation of interacting fluids, we have

tested two configurations. We call it *medium* and *large*. The circle dataset was homogeneous, the vorticy dataset has areas with more slope and with less, comparisons between sets of isocontours cannot be made on equal footing. Our *large* isocontour range has its first isovalue at 5.5278, contains 9 isocontours with an isovalue increment of 1.0 for each isocontour. The *medium* dataset starts at the same isovalue, but the increment is 0.5 in this case. We achieve a total compression of 16.6 bits per vertex on the *large* isocontour range and 12.08 for the medium isocontour range. The bits per vertex we obtain for a single isosurface is 33.97, which tells of the harshness of the dataset. Using extrapolating predictors with a reduced number of known points (close to 4 on our 2D tests) is ineffective for the lossless compression of floating point data. This causes us to achieve subobtimal results in single isocontour transmission. The following isocontour transmission will reduce the average bit per vertex.

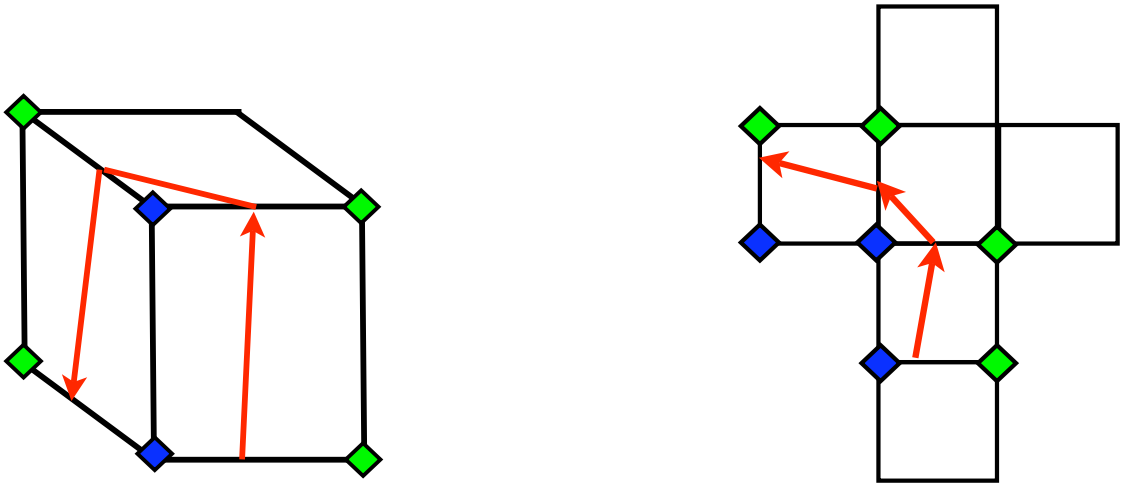
We have tested encoding the previous set of isocontours in different orders (see Table 8). In all cases we have observed that if the objective is to encode all the nodes in a range, it is an improvement encoding them sequentially rather than hierarchically. In the sequential encoding we have appreciated a 20% improvement over hierarchical encoding. The improvement is dependent on the accuracy of prediction. Sequential encoding encodes each point with a close-to-full stencil (7 to 8 neighbors), while hierarchical encoding is relegated to use stencils scarcely populated ((4 neighbors on average) for half the encoded isocontours.

Table 8: Total compressed files for the same set of isocontours for the 2D circle, in different encoding order.

	Total File Size
Medium {1,2,3,4,5,6,7,8,9}	32537
Medium {1,3,5,7,9,2,4,6,8}	41214
Medium {1,5,9,3,7,2,4,6,8}	44599

8.3 Extension to 3D

We have extended our 2D approach to the transmission and compression of isosets in 3D. There are a few differences between the 2D and the 3D implementations, manly to address the complexity of the extraction, traversal and encoding of the isoset. We discuss these changes and provide results for the 3D version of our approach.



(a) 3D cell *(b) Unfolded 3D cell*

Figure 67: The intersection between the faces of a 3D cell and the isoset form a series of closed curves on the surface of the cell, represented in red. Blue nodes are below the isovalue, and green nodes are above it.

The encoding of an isocontour in 2D started with a set of seeds, one for each connected component. This has not changed in the 3D case, but the definition of a seed has. In 3D, a seed is stick in the 3D regular grid.

Isosurfaces can be extracted by continuation [108]: starting from a seed cell in an isosurface component, the entire connected component can be traversed using breath-first traversal through adjacent cells. Our traversal method bears strong resemblance with established methods such as Mascarenhas et al [75].

Our traversal is described in the pseudocode of Listing 8.1. We process the cells in breadth-first order. For each cell, we process the intersection of the isosurface with the cell. There can be more than one intersection between the isosurface and a cell, for that reason we do not mark a cell, but for each cell we mark the visited edges.

Listing 8.1: Traversal pseudocode for a connected component of an isosurface.

```
TraverseComponent(Cell c)
    ListOfCells lc;
    Queue Q;
    Enqueue(Q, c); // We encode the combination of cell and edge
    Mark(c);     // For each cell, we mark the set of used edges
    while (Q not empty)
    {
        u = Dequeue(Q);
        lc = ProcessCell(u);
        for each cell v on lc
        {
            if (v not marked)
            {
                Enqueue(Q, v);
                Mark(v);
            }
        }
    }
```

In 3D an isoset is a surface. It intersects cells (cubes) by creating vertices on the cell's edges, and creates edges on the cell's faces. The set of edges of an isocontour that lie in a cell form a set of connected loops (see Figure 67), see also [6].

The entrance to a cell in 3D is defined by two sticks that identify an edge of the isocontour. Using the property that all isocontour's edges are connected in a loop, we can reduce the dimensionality of the problem. We pick one of the two entrance sticks, and the face that is not shared with the remaining entrance stick. The combination of an entrance stick with a face (a 2D square, same as a 2D cell) is perfect for the algorithm of processing of a cell in 2D.

In 2D, each cell has an entrance stick, it computes the exit stick and encodes any missing node values (between zero and two). In 3D, there is no enter stick, the entrance point is a face, represented as two sticks. Between the two sticks there is an edge of the isocontour mesh. This edge from the isocontour is analog to the 2D isocurve, we apply the 2D algorithm that follows isocurves through 2D cells (squares) to this analogy; the result is a traversal of the faces of the 3D cell that end up at the beginning again. Listing 8.2 details our algorithm to process a 3D cell. We mark the input edge on the cell as visited. We find one of the faces

Listing 8.2: Pseudocode that processes a 3D cell.

```
ProcessCell(Cell c, Edge e)
  MarkEdge(c, e);
  f = FindFace(c, e); // we locate a face that contains the edge
  // and process the face as if it were a 2D cell :
  <f, e> = ProcessCell2D(f, e);
  while (<c,e> not marked)
  {  // keep traversing the same cell until the marked edge
    MarkEdge(c, e);
    <f,e> = ProcessCell2D(f,e);
  }
```

containing the edge and perform the 2D cell processing. The 2D cell processing encodes the values at nodes that are needed and not known, and returns the next face and edge of the cell to be processed.

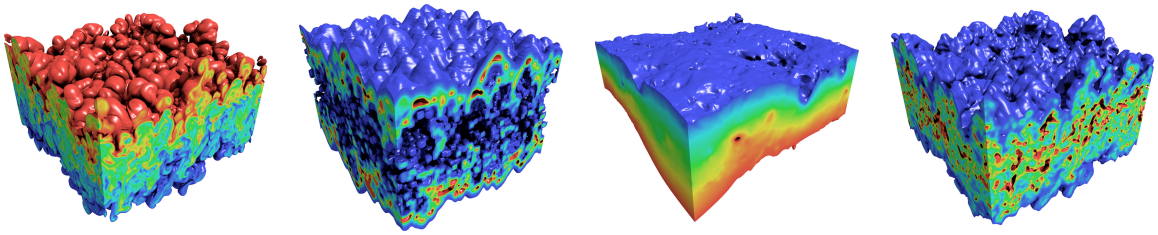


Figure 68: Volume rendering from Scientific datasets at LLNL [23], used to extract ranges of isosets. They are respectively: density, diffusivity, pressure and viscocity.

We have implemented prediction of missing values of the 3D grid by using 2D Spectral predictions. We compute 3 2D Spectral predictions, and we chose the prediction with more known values in the stencil. We consider all axis-aligned 2D 3×3 stencils containing the unknown value and select the set with the largest number of known values. Remember that Section 4.4.5.2 shows that stencils with more known values are often more accurate than stencils with fewer known values. If several stencils have the same number of known values, we combine their predictors using the mean or the median of these values. The mean is preferable for very smooth data. The mean has the advantage of being a single Spectral predictor of a 3×3 stencil, and as thus more resistant to noise than the mean.

It is important to note that our scheme encodes the values at the nodes that define the isoset. The client decodes the nodes, and extracts the isosurface from the reduced set

of nodes. The isosurface extraction is completely independent of our traversal, which only encodes the nodes and makes no implications on the topology or connectivity.

Table 9: Results for the 3D encoding of ranges of isosets for the Miranda data.

Dataset	Start Isovalue	Bits per triangle (1 isoset)	Bits per triangle (range)
density	1.734	12.609	6.824
diffusivity	0.100	13.985	9.783
pressure	100.274	4.228	0.995
viscocity	0.456	12.823	8.246

We have tested our results in a 3D distance field. A single isoset compression resulted in 9.4 bits per node, or equivalently transmitted 5.23 bits per triangle of the isoset. We tested two families of isosets, one with an isostep value increment of 2.0 obtaining 4.1 bits per triangle, and one with an isostep increment of 1.0 resulting in 1.4 bits per triangle.

As described in Table 9, we have compressed several ranges of isosets on data from the Miranda simulation at LLNL [23]. We achieve results (counting connectivity and geometry) between 7 and 9 bits per triangle, without any quantization of the geometry. The datasets we tested can be seen in Figure 68.

8.4 Conclusion

We have presented a technique for the interactive exploration of scalar fields in 2D and 3D. Our method is best suited for exploring a range of contiguous isocontours, in which case the isocontours should be encoded sequentially. Our method requires a set of seeds as input. The prediction used in 3D is an extension from 2D prediction, there is room for improvement.

The method is applicable to remote clients, only the nodes that determine an isocontour range are transmitted, clients can browse large isocontours using memory footprint relative to the size of the isocontour, instead of the scalar field.

CHAPTER IX

CONCLUSION

In this thesis we have presented several approaches for the compression of graphical models. Our methods are based on predictive compression, taking advantage of the inter-dimensional coherence present in scalar fields, animated meshes and isosets.

For compression of scalar fields sampled in regular grids, we have introduced the Lorenzo Predictor, optimal predictor in a n-dimensional cube. Its extensions to cubes of the form 3^n, the Radial and the bi-Lorenzian predictors, are able to predict smooth datasets better than the Lorenzo predictor. We have proposed the Spectral family of predictors, a method to generate the optimal predictor for any neighborhood. The set of Spectral predictors can be precomputed and stored in a table. We have implemented hierarchical techniques that make use of the spectral predictors. Extending the Spectral predictors to 3D is a venue for future work.

For compression of animated triangle meshes, we have proposed Dynapack, which extends the Lorenzo predictor to triangle meshes and combines it with the Edgebreaker connectivity compression to obtain an efficient lossless compression technique for sequences of triangle meshes with constant connectivity. We have extended Dynapack to support lossy compression by introducing Clippacker. Clippacker segments the animation into clips; each clip is simplified, and compressed with Dynapack. Our preliminary results with already simplified meshes were not encouraging; Clippacker is geared towards animation with high sampling rates.

We have proposed a novel method for the encoding of the simplification process in a triangle mesh. The upper bound cost of compression is 1.2 bits per triangle, but the technique requires to have the full unsimplified connectivity. This encoding can be used for geomorphs or for the transmission and speed up of view dependent simplification applications. This approach opens a new area of possibilities, which to our knowledge has not yet

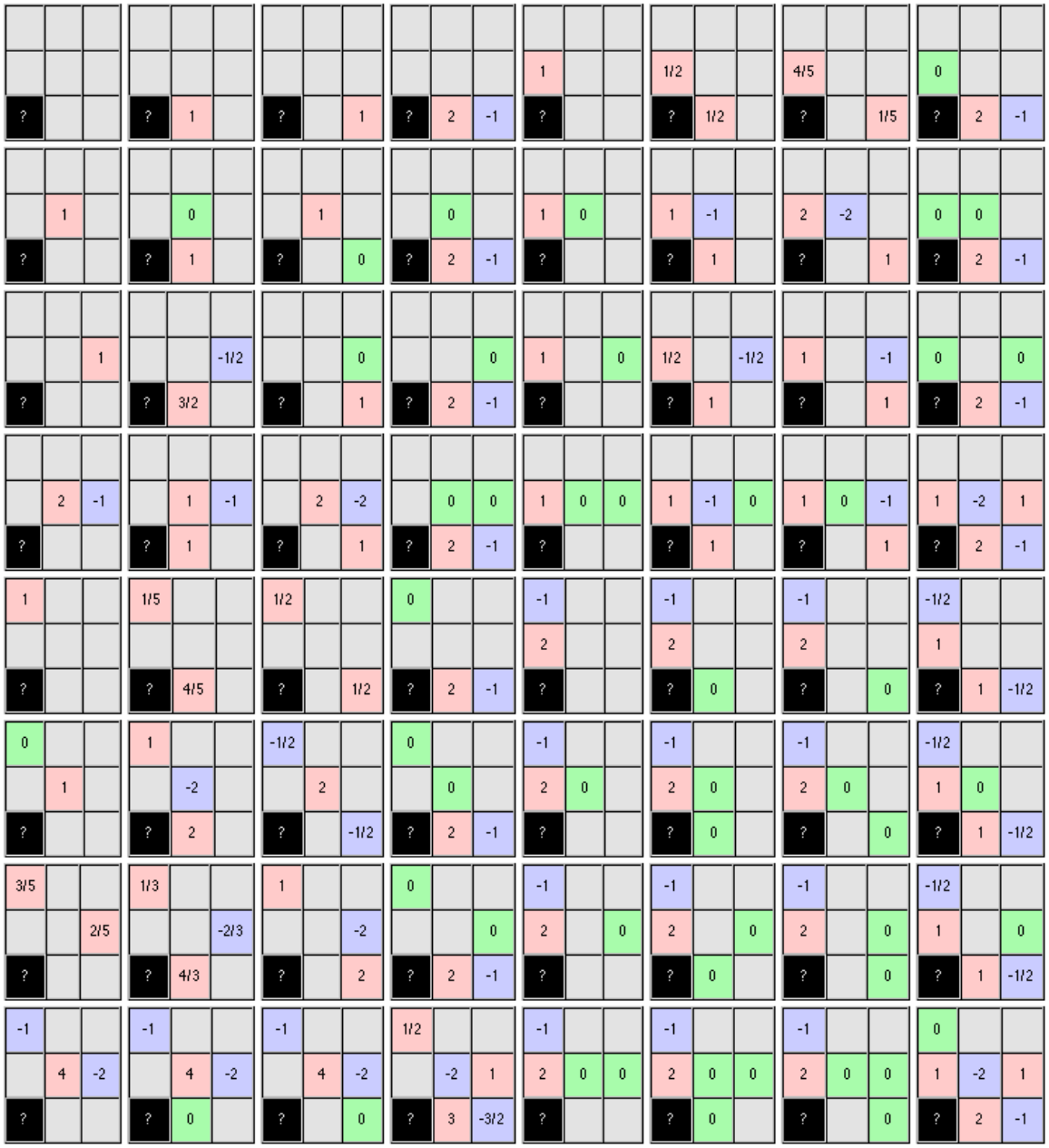
been explored.

For the compression and exploration of isosets with varying isovalues, we have presented our approach that compresses the set of values from the regular grid; the set of values identifies the different isosets. Through the compression of values, subsequent isosets can use previously encoded values, i.e. a value is not transmitted twice. Encoded nearby values are used in the prediction of new values. Our approach extends to 2D and 3D. Our compression performs optimally when a section of contiguous isosets is sent in progressive sequence.

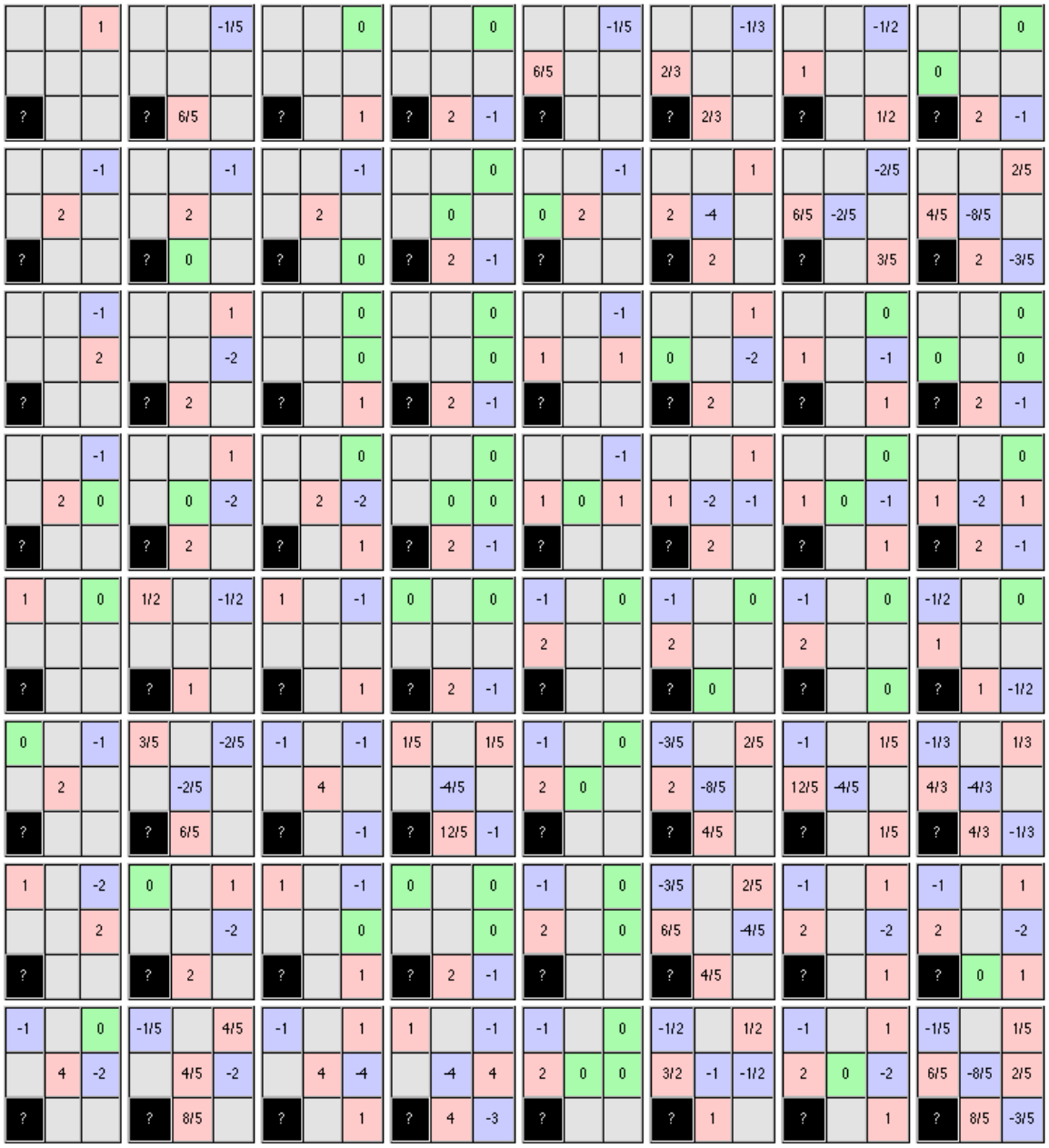
APPENDIX A

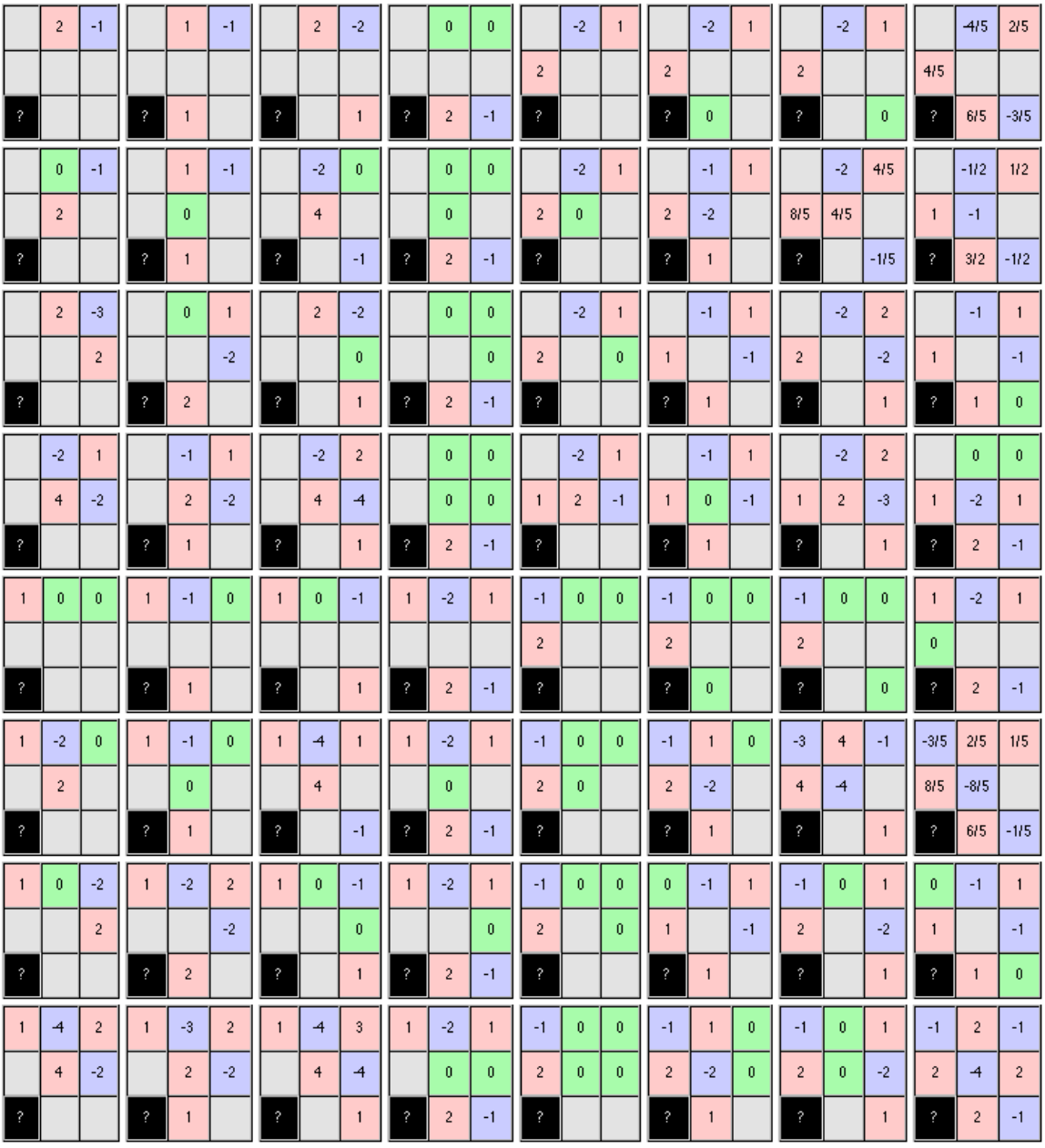
SPECTRAL TABLE

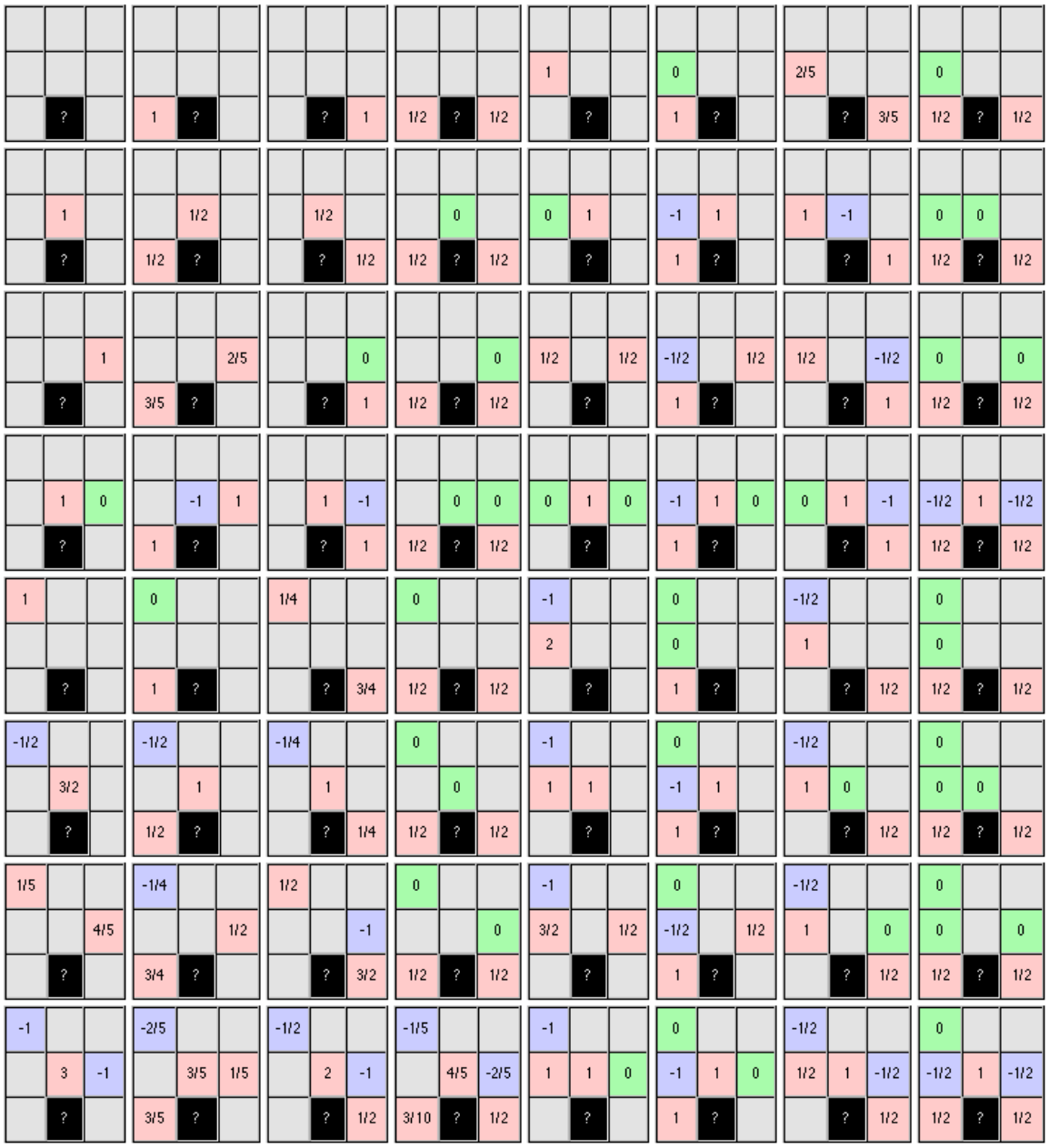
We present the computed Spectral Predictors for a 3×3 neighborhood. The '?' marks the position of the predicted. These are the weights for the corner, edge and face cases.

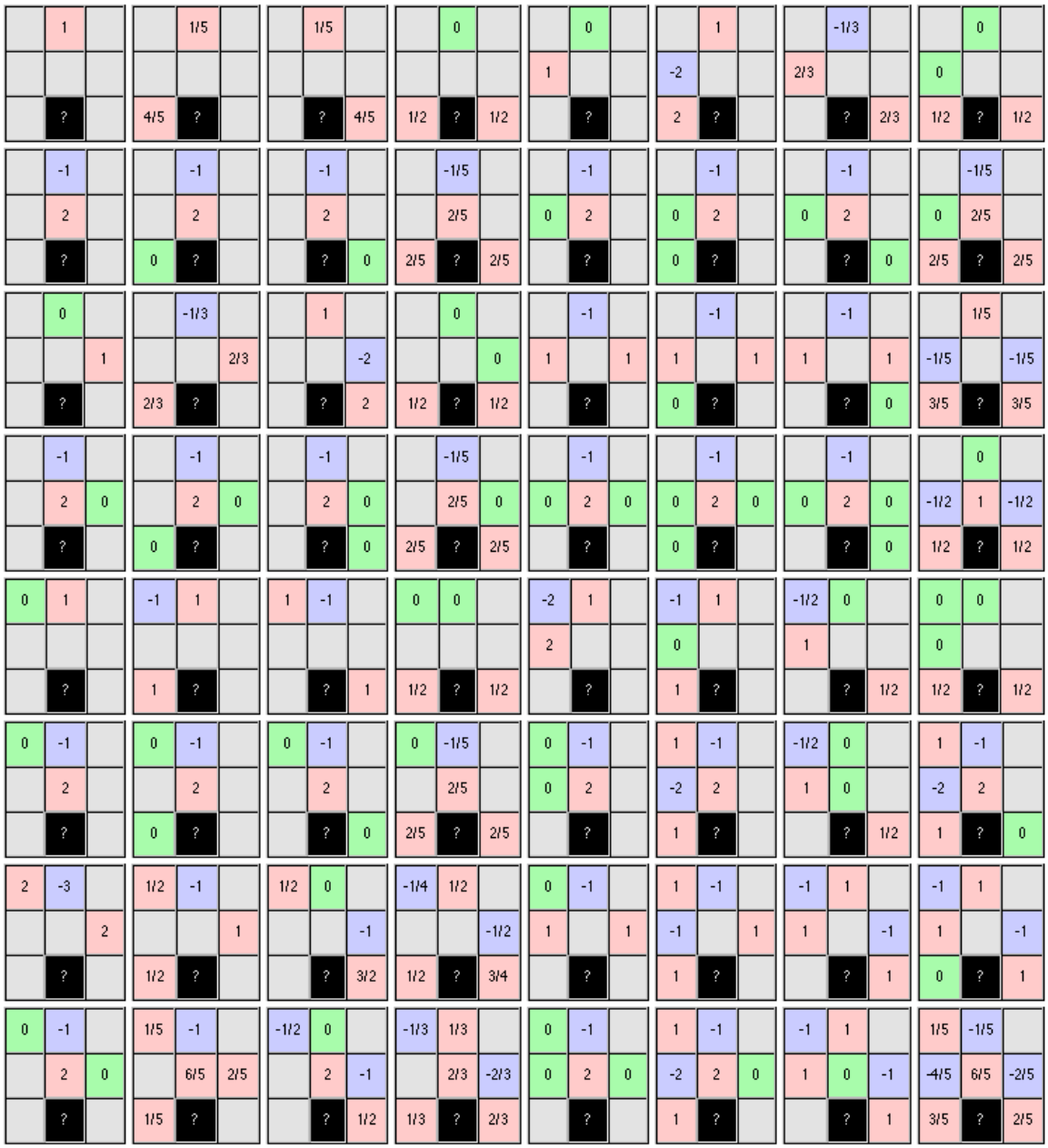


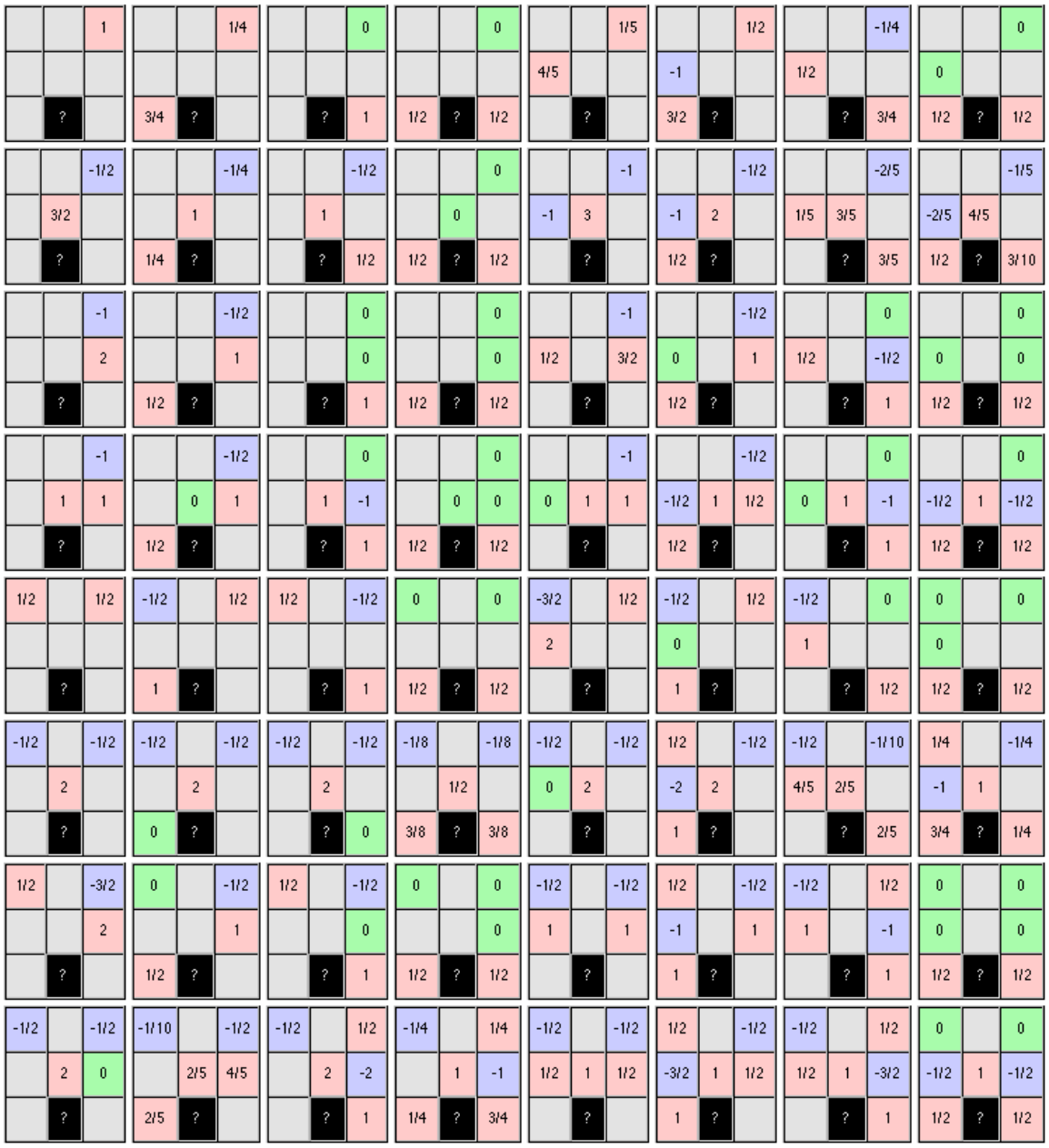

1
0
2/5
0
-1/2
-1/2
-2/3
0
3/2
1
4/3
0
?
?
1
?
3/5
?
2
-1
?
1/2
?
1/3
?
2
-1
-1
0
-2
0
-1
0
-2
0
2
0
4
0
1
1
1
-1
0
4
0
0
?
?
1
?
-1
?
2
-1
?
1
?
-1
?
2
-1
1/2
1/2
2
0
-1
-1/4
-2
-1/2
1/2
-1
-4
0
3/2
1/2
3/4
-1/4
2
2
1/2
1/2
?
3/2
?
3
?
2
-1
?
?
?
3/4
?
?
-1
?
3/2
-1
-1
0
-2
0
-1
0
-2
0
3
-1
1
-1
4
0
0
0
1
1
0
1
-1
0
1
2
1
1
-2
1
?
?
1
?
-1
?
2
-1
?
1
?
?
?
2
-1
1
0
1
-1
2
-2
0
0
0
-1
0
-1
0
-1
0
-1/2
0
2
2
2
1
?
?
1
?
1
?
2
-1
?
0
?
0
?
1
-1/2
1
-2
1
-1
2
-2
0
0
-1
0
-1
1
-3/2
1
-1
1
0
4
0
0
2
0
2
-2
3
-2
2
-2
?
1
?
-1
?
2
-1
?
?
?
1
?
1/2
?
1
0
3
-4
-1
2
3/2
-1
1/2
-1
-1
0
-1
1/2
-3/2
1
-1/2
0
2
-2
1
2
0
3/2
-1/2
2
-1
1
0
?
2
?
-1
?
1
-1
?
1/2
?
3/2
?
2
-3/2
?
?
?
1/2
?
1
-1/2
-1
0
-1
1
-1/2
-1
1
-1
0
-1
1
-2
2
4
-2
2
-2
4
-1
-2
2
2
0
0
2
-2
0
3
-2
-1
-1/2
1/2
?
1
?
-1/2
?
3
-2
?
?
?
1
?
3/2
-1/2

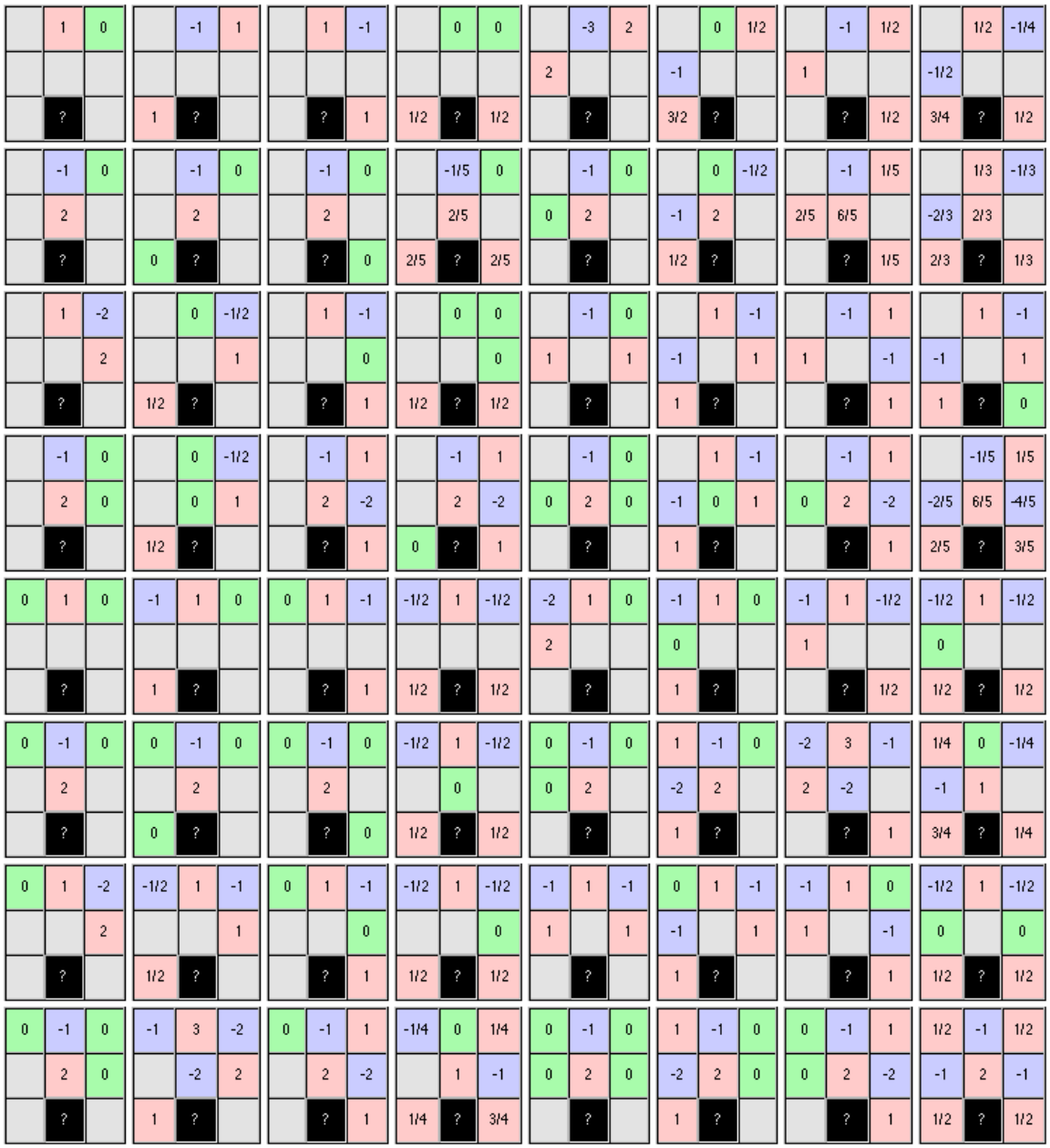

1 -1/5 0 0 -1/5 -1/3 -1/2 0
6/5 2/3 1 0
? ? ? ? 6/5 ? 1 ? 2 -1 ? ? ? 2/3 ? ? 1/2 ? 2 -1
-1 -1 -1 0 -1 1 -2/5 2/5
2 2 2 0 0 2 2 -4 6/5 -2/5 4/5 -8/5
? ? ? 0 ? 0 ? 2 -1 ? ? 2 ? 3/5 ? 2 -3/5
-1 1 0 0 -1 1 0 1 0
2 -2 0 0 1 1 0 -2 1 -1 0 0
? ? ? 2 ? 1 ? 2 -1 ? ? 2 ? 1 ? 2 -1
-1 1 0 0 -1 1 0 1 0
2 0 0 -2 2 -2 0 0 1 0 1 1 -2 -1 1 0 -1 1 -2 1
? ? ? 2 ? 1 ? 2 -1 ? ? 2 ? 1 ? 2 -1
1 0 1/2 -1/2 1 -1 0 0 -1 0 -1 0 -1 0 -1/2 0
2 2 2 1
? ? ? 1 ? 1 ? 2 -1 ? ? 0 ? 0 ? 1 -1/2
0 -1 3/5 -2/5 -1 -1 1/5 1/5 -1 0 -3/5 2/5 -1 1/5 -1/3 1/3
2 -2/5 4 -4/5 2 0 2 -8/5 12/5 -4/5 4/3 -4/3
? ? ? 6/5 ? -1 ? 12/5 -1 ? ? ? 4/5 ? ? 1/5 ? 4/3 -1/3
1 -2 0 1 1 -1 0 0 -1 0 -3/5 2/5 -1 1 -1 1
2 2 0 0 2 0 6/5 -4/5 2 -2 2 -2
? ? ? 2 ? 1 ? 2 -1 ? ? ? 4/5 ? ? 1 ? 0 1
-1 0 -1/5 4/5 -1 1 1 -1 -1 0 -1/2 1/2 -1 1 -1/5 1/5
4 -2 4/5 -2 4 -4 -4 4 2 0 0 3/2 -1 -1/2 2 0 -2 6/5 -8/5 2/5
? ? ? 8/5 ? 1 ? 4 -3 ? ? ? 1 ? ? 1 ? 8/5 -3/5


2 -1
?
1 -1
? 1
2 -2
? 1
0 0
? 2 -1
-2 1
2
?
-2 1
2
? 0
-2 1
2
?
-4/5 2/5
4/5
? 6/5 -3/5
0 -1
2
?
1 -1
0
? 1
-2 0
4
? -1
0 0
0
? 2 -1
-2 1
2 0
?
-1 1
2 -2
? 1
-2 4/5
8/5 4/5
? -1/5
-1/2 1/2
1 -1
? 3/2 -1/2
2 -3
2
?
0 1
-2
? 2
2 -2
0
? 1
0 0
0
? 2 -1
-2 1
2 0
?
-1 1
1 -1
? 1
-2 2
2 -2
? 1
-1 1
1
? 1 0
-2 1
4 -2
?
-1 1
2 -2
? 1
-2 2
4 -4
? 1
0 0
0 0
? 2 -1
-2 1
1 2 -1
?
-1 1
1 0 -1
? 1
-2 2
1 2 -3
?
0 0
1 -2 1
? 2 -1
1 0 0
?
1 -1 0
? 1
1 0 -1
? 1
1 -2 1
? 2 -1
-1 0 0
2
?
-1 0 0
2
? 0
-1 0 0
2
? 0
1 -2 1
0
? 2 -1
1 -2 0
2
?
1 -1 0
0
? 1
1 -4 1
4
? -1
1 -2 1
0
? 2 -1
-1 0 0
2 0
?
-1 1 0
2 -2
? 1
-3 4 -1
4 -4
? 1
-3/5 2/5 1/5
8/5 -8/5
? 6/5 -1/5
1 0 -2
2
?
1 -2 2
-2
? 2
1 0 -1
0
? 1
1 -2 1
0
? 2 -1
-1 0 0
2 0
?
0 -1 1
1 -1
? 1
-1 0 1
2 -2
? 1
0 -1 1
1
? 1 0
1 -4 2
4 -2
?
1 -3 2
2 -2
? 1
1 -4 3
4 -4
? 1
1 -2 1
0 0
? 2 -1
-1 0 0
2 0 0
?
-1 1 0
2 -2 0
? 1
-1 0 1
2 0 -2
? 1
-1 2 -1
2 -4 2
? 2 -1

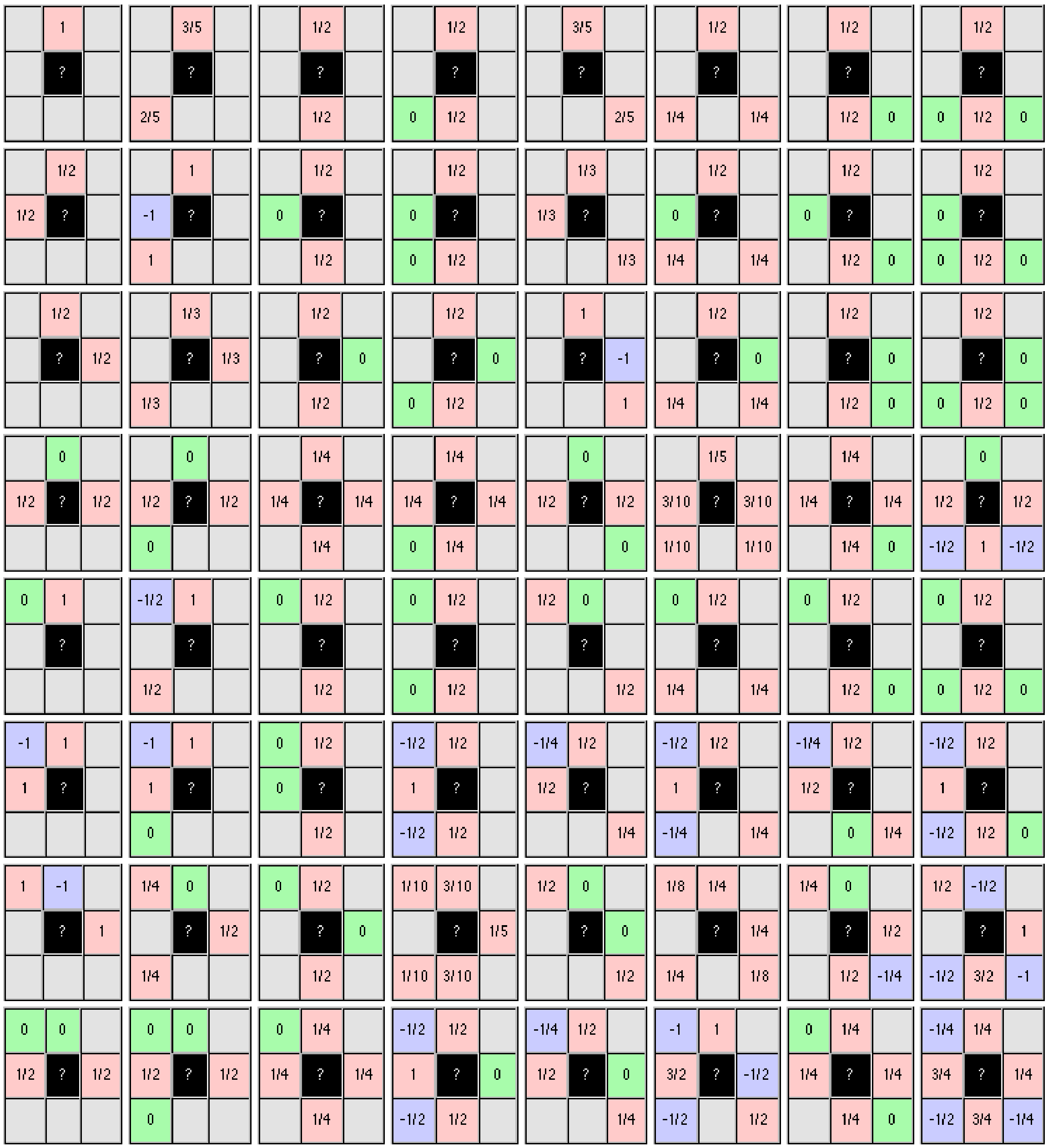

? 1 ? ? 1 1/2 ? 1/2 1 ? 0 1 ? 2/5 ? 3/5 0 1/2 ? 1/2
1 1/2 1/2 ? 1/2 ? 1/2 0 ? 1/2 0 1 ? -1 1 ? 1 -1 ? 1 0 0 1/2 ? 1/2
1 2/5 3/5 ? 0 ? 1 0 1/2 ? 1/2 1/2 ? 1/2 -1/2 1 ? 1/2 1/2 -1/2 ? 1 0 0 1/2 ? 1/2
1 0 -1 1 1 ? 1 -1 ? 1 0 0 1/2 ? 1/2 0 1 0 ? -1 1 0 0 1 -1 ? 1 -1/2 1 -1/2 1/2 ? 1/2
1 0 1/4 0 -1 2 0 1 ? 0 ? 3/4 1/2 ? 1/2 1 ? 0 -1/2 1 ? 0 0 1/2 ? 1/2
-1/2 -1/2 -1/4 0 -1 1 0 1 1 ? 1 ? 0 -1 1 ? -1/2 1 0 ? 1/2 0 0 1/2 ? 1/2
1/5 4/5 -1/4 1/2 1/2 -1 0 ? 3/4 ? 3/2 1/2 ? 1/2 -1 3/2 1/2 ? 0 -1/2 1/2 ? -1/2 1 0 ? 1/2 0 0 1/2 ? 1/2
-1 3 -1 -2/5 3/5 1/5 -1/2 2 -1 3/5 ? 1/2 -1/5 4/5 -2/5 3/10 ? 1/2 -1 1 1 0 ? 0 -1 1 0 1 ? -1/2 1/2 1 -1/2 ? 1/2 0 -1/2 1 -1/2 1/2 ? 1/2


1
?
1/5
4/5
?
1/5
?
4/5
0
1/2
?
1/2
0
1
?
1
-2
2
?
-1/3
2/3
?
0
0
?
1/2
?
1/2
-1
2
?
-1
2
0
?
-1
2
?
0
-1/5
2/5
2/5
?
2/5
-1
0
2
?
-1
0
2
?
-1
0
2
?
-1/5
0
2/5
2/5
?
2/5
0
1
?
-1/3
2/3
2/3
?
1
-2
?
2
0
1/2
?
1/2
-1
1
1
?
-1
1
1
?
-1
1
?
1/5
-1/5
3/5
?
3/5
-1
2
0
?
-1
2
0
?
-1
2
0
?
-1/5
2/5
0
2/5
?
2/5
-1
0
2
0
?
-1
0
2
0
?
-1
0
2
0
?
0
-1/2
1
-1/2
1/2
?
1/2
0
1
?
-1
1
1
?
1
-1
?
0
0
1/2
?
1/2
-2
1
2
?
-1
1
0
?
-1/2
0
1
?
0
0
0
1/2
?
1/2
0
-1
2
?
0
-1
2
0
?
0
-1/5
2/5
2/5
?
2/5
0
-1
0
2
?
1
-1
-2
2
1
?
-1/2
0
1
0
?
1/2
1
-1
-2
2
1
?
0
2
-3
?
2
1/2
-1
1/2
?
1
-1/4
1/2
1/2
?
-1/2
0
-1
1
?
3/2
1/2
?
3/4
0
-1
1
?
1
-1
-1
1
?
1
-1
1
?
-1
1
1
?
-1
1
?
1
0
?
1
0
-1
2
?
1/5
-1
6/5
2/5
1/5
?
-1/2
0
2
-1
?
1/2
-1/3
1/3
2/3
-2/3
1/3
?
2/3
0
-1
0
2
0
?
1
-1
-2
2
0
1
?
-1
1
1
0
-1
?
1
1/5
-1/5
-4/5
6/5
-2/5
3/5
?
2/5

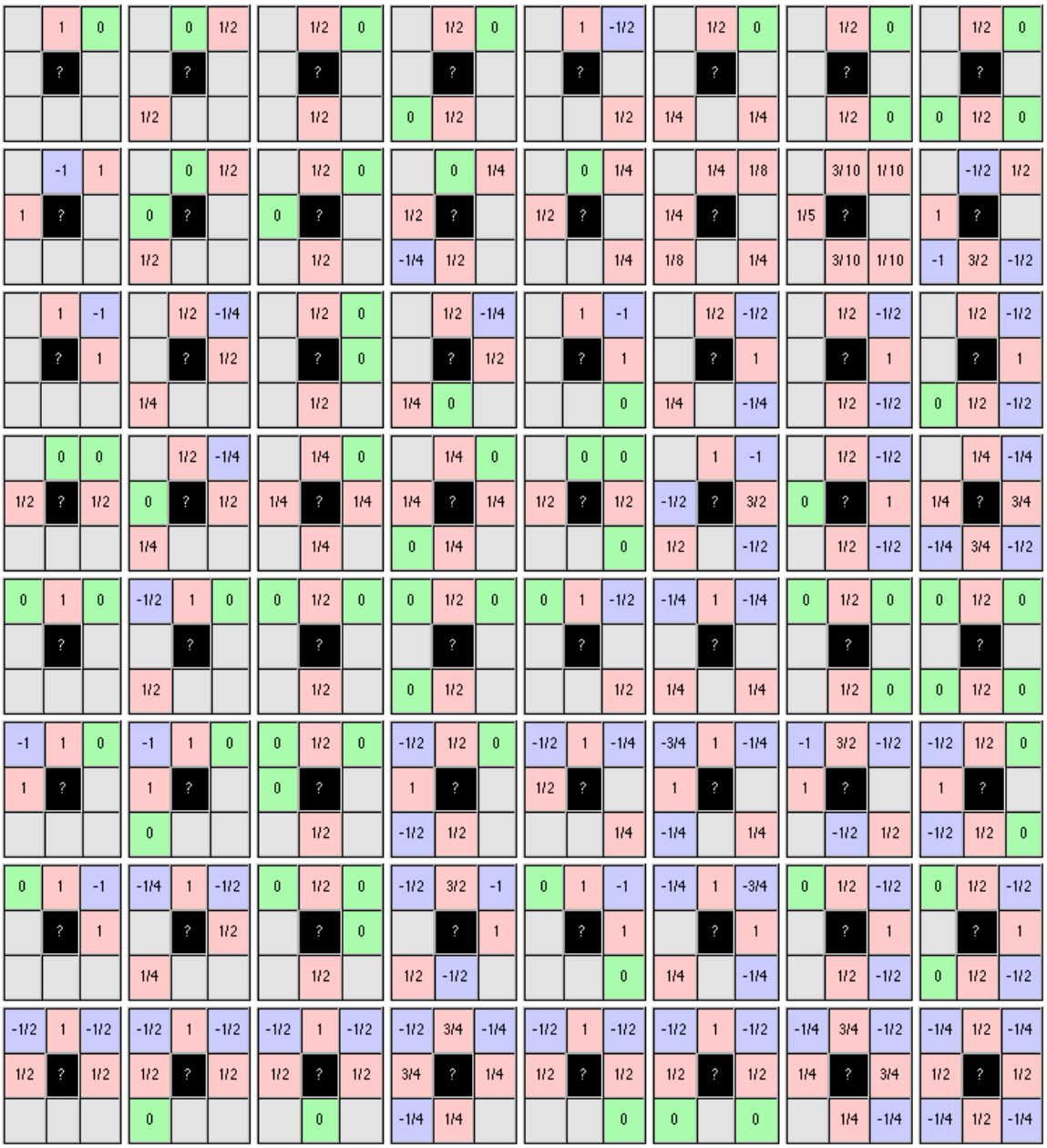

1 1/4 0 0 1/5 1/2 -1/4 0
? 3/4 ? ? 1 1/2 ? 1/2 4/5 ? -1 ? 3/2 ? 1/2 ? 3/4 0 ? 1/2 ? 1/2
-1/2 -1/4 -1/2 0 -1 -1/2 -2/5 -1/5
3/2 1 1 0 -1 3 -1 2 1/5 3/5 -2/5 4/5
? 1/4 ? ? 1 1/2 1/2 ? 1/2 ? ? 1/2 ? ? 3/5 1/2 ? 3/10
-1 -1/2 0 0 -1 -1/2 0 0
2 1 0 0 1/2 3/2 0 1 1/2 -1/2 0 0
? 1/2 ? ? 1 1/2 ? 1/2 ? 1/2 ? 1/2 ? ? 1 1/2 ? 1/2
-1 -1/2 0 0 -1 -1/2 0 0
1 1 0 1 1 -1 0 0 0 1 1 -1/2 1 1/2 0 1 -1 -1/2 1 -1/2
? 1/2 ? ? 1 -1 1/2 ? 1/2 ? ? 1/2 ? ? 1 1/2 ? 1/2
1/2 1/2 -1/2 1/2 1/2 -1/2 0 0 -3/2 1/2 -1/2 1/2 -1/2 0 0 0
? 1 ? ? ? 1 1/2 ? 1/2 2 ? 0 ? 1 ? 1/2 0 ? 1/2 ? 1/2
-1/2 -1/2 -1/2 -1/2 -1/2 -1/2 -1/8 -1/8 -1/2 -1/2 1/2 -1/2 -1/2 -1/10 1/4 -1/4
2 2 2 1/2 0 2 -2 2 4/5 2/5 -1 1
? 0 ? ? ? 0 3/8 ? 3/8 ? ? 1 ? ? ? 2/5 3/4 ? 1/4
1/2 -3/2 0 -1/2 1/2 -1/2 0 0 -1/2 -1/2 1/2 -1/2 -1/2 1/2 0 0
2 1 0 0 1 1 -1 1 1 -1 0 0
? 1/2 ? ? ? 1 1/2 ? 1/2 ? ? 1 ? 1 ? ? 1 1/2 ? 1/2
-1/2 -1/2 -1/10 -1/2 -1/2 1/2 -1/4 1/4 -1/2 -1/2 1/2 -1/2 -1/2 1/2 0 0
2 0 2/5 4/5 2 -2 1 -1 1/2 1 1/2 -3/2 1 1/2 1/2 -3/2 -1/2 1 -1/2
? 2/5 ? ? ? 1 1/4 ? 3/4 ? ? 1 ? ? ? 1 1/2 ? 1/2


1 0 ? -1 1 1 ? 1 -1 ? 1 0 0 1/2 ? 1/2 -3 2 2 ? 0 1/2 -1 3/2 ? -1 1/2 1 ? 1/2 1/2 -1/4 -1/2 3/4 ? 1/2
-1 0 2 ? -1 0 2 0 ? -1 0 2 ? 0 -1/5 0 2/5 2/5 ? 2/5 -1 0 0 2 ? 0 -1/2 -1 2 1/2 ? -1 1/5 2/5 6/5 ? 1/5 1/3 -1/3 -2/3 2/3 2/3 ? 1/3
1 -2 2 ? 0 -1/2 1 1/2 ? 1 -1 0 1/2 ? 0 0 0 1/2 ? 1/2 -1 0 1 ? 1 -1 1 -1 ? 1 -1 1 -1 ? 1 1 ? 0
-1 0 2 0 ? 0 -1/2 0 1 1/2 ? -1 1 2 -2 ? 1 -1 2 -2 0 ? 1 -1 0 0 2 0 ? 1 -1 -1 0 1 ? -1 1 0 2 -2 ? 1 -1/5 1/5 -2/5 6/5 -4/5 2/5 ? 3/5
0 1 0 ? -1 1 0 1 ? 0 1 -1 ? -1/2 1 -1/2 1/2 ? 1/2 -2 1 0 2 ? 0 -1 1 0 1 ? 0 -1 1 -1/2 1 ? 1/2 -1/2 1 -1/2 0 1/2 ? 1/2
0 -1 0 2 ? 0 -1 0 2 0 ? -1 0 2 ? 0 -1/2 1 -1/2 1/2 ? 1/2 0 -1 0 0 2 ? 0 1 -1 0 -2 2 1 ? -2 3 -1 2 -2 ? 1 1/4 0 -1/4 -1 1 3/4 ? 1/4
0 1 -2 2 ? -1/2 1 -1 1/2 ? 0 1 -1 0 ? -1/2 1 -1/2 1/2 ? 1/2 -1 1 -1 1 1 ? 0 1 -1 -1 0 1 ? -1 1 0 1 -1 ? 1 -1/2 1 -1/2 0 0 1/2 ? 1/2
0 -1 0 2 0 ? -1 3 -2 -2 2 1 ? 0 -1 1 2 -2 ? 1 -1/4 0 1/4 1 -1 1/4 ? 3/4 0 -1 0 0 2 0 ? 1 -1 0 -2 2 0 ? 0 -1 1 0 2 -2 ? 1 1/2 -1 1/2 -1 2 -1 1/2 ? 1/2


? 1 ? 1 ? 1 ? 0 1 ? 1 ? 1/2 1/2 ? 1 0 ? 0 1 0
1 ? 0 1 ? 0 1/2 ? 1/2 1 ? -1 1 3/5 ? 2/5 1 ? -1/2 1/2 1 ? -1 1 ? -1 1 0
? 1 2/5 ? 3/5 1/2 ? 1/2 ? 1 1 -1 ? 1 0 ? 1/2 -1/2 ? 1 -1 ? 0 1 -1
1/2 ? 1/2 0 0 0 0 0 0 0 0 0 0 0 -1/2 1 -1/2
1 ? 1/2 1/2 ? 2/5 3/5 1/2 ? -1/2 1 ? 1/2 ? 0 1/2 ? 1/2 ? 0 1/2 ? -1/2 1 0
0 1 ? 0 1 ? 0 -1 ? 1 1 ? 0 1/2 ? 0 1 ? -1 1 ? 0 1/2 ? 0 1 ? -1 1 0
2/5 ? 3/5 1/4 ? 1/2 1/4 1/3 ? 1/3 1/4 ? 1/2 1/4 1/2 ? 0 1/2 1/4 ? 1/2 1/4 ? 1/2 1/2 -1/4 -1/4 1 -1/2
0 1/2 ? 1/2 0 0 1/2 ? 1/2 0 0 0 0 0 0 0 0 0 0 1/4 0 ? 1/2 1/2 -1/4 0 1/2 ? 1/2 -1/2 1 -1/2


1 ?
3/5 ? 2/5
1/2 ? 1/2
1/2 ? 0 1/2
3/5 ? 2/5
1/2 ? 1/4 1/4
1/2 ? 1/2 0
1/2 ? 0 1/2 0
1/2 1/2 ?
1 -1 ? 1
1/2 0 ? 1/2
1/2 0 ? 0 1/2
1/3 1/3 ? 1/3
1/2 0 ? 1/4 1/4
1/2 0 ? 1/2 0
1/2 0 ? 0 1/2 0
1/2 ? 1/2
1/3 ? 1/3 1/3
1/2 ? 0 1/2
1/2 ? 0 0 1/2
1 ? -1 1
1/2 ? 0 1/4 1/4
1/2 ? 0 1/2 0
1/2 ? 0 0 1/2 0
0 1/2 ? 1/2
0 1/2 ? 1/2 0
1/4 1/4 ? 1/4 1/4
1/4 1/4 ? 1/4 0 1/4
0 1/2 ? 1/2 0
1/5 3/10 ? 3/10 1/10 1/10
1/4 1/4 ? 1/4 1/4 0
0 1/2 ? 1/2 -1/2 1 -1/2
0 1 ?
-1/2 1 ? 1/2
0 1/2 ? 1/2
0 1/2 ? 0 1/2
1/2 0 ? 1/2
0 1/2 ? 1/4 1/4
0 1/2 ? 1/2 0
0 1/2 ? 0 1/2 0
-1 1 1 ?
-1 1 1 ? 0
0 1/2 0 ? 1/2
-1/2 1/2 1 ? -1/2 1/2
-1/4 1/2 1/2 ? 1/4
-1/2 1/2 1 ? -1/4 1/4
-1/4 1/2 1/2 ? 0 1/4
-1/2 1/2 1 ? -1/2 1/2 0
1 -1 ? 1
1/4 0 ? 1/2 1/4
0 1/2 ? 0 1/2
1/10 3/10 ? 1/5 1/10 3/10
1/2 0 ? 0 1/2
1/8 1/4 ? 1/4 1/4 1/8
1/4 0 ? 1/2 1/2 -1/4
1/2 -1/2 ? 1 -1/2 3/2 -1
0 0 1/2 ? 1/2
0 0 1/2 ? 1/2 0
0 1/4 1/4 ? 1/4 1/4
-1/2 1/2 1 ? 0 -1/2 1/2
-1/4 1/2 1/2 ? 0 1/4
-1 1 3/2 ? -1/2 -1/2 1/2
0 1/4 1/4 ? 1/4 1/4 0
-1/4 1/4 3/4 ? 1/4 -1/2 3/4 -1/4


1 1/2 2/5 1/2 1/2 1/2 1/2 1/2
? ? ? ? ? ? ? ?
1/2 3/5 1/2 0 1/2 0 1 -1/2 0 1 -1/2
2/5 1/2 1/3 1/4 1/4 1/4 1/4 1/4
3/5 ? 0 ? 1/3 ? 1/2 ? 1/2 ? 1/2 ? 1/2 ? 1/2 ?
1/2 1/3 -1/4 1/2 1/4 0 1/4 0 1/4 -1/2 1 -1/4
0 1/2 1 1/2 0 0 0 0
? 1 ? 0 ? -1 ? 0 ? 1 ? 1 ? 1 ? 1
1/2 1/2 0 1/2 0 1/2 -1/2 1 -1 0 1 -1
0 0 0 1/4 0 0 0 0
1/2 ? 1/2 1/2 ? 1/2 1/2 ? 1/2 0 1/2 ? 1/2 1/2 ? 1/2 1/2 ? 1/2 1/2 ? 1/2 1/2 ? 1/2
0 0 -1/4 1/2 0 0 0 -1/2 1 -1/2
1/2 1/2 0 1/2 1/4 1/4 1/4 1/4 0 1/4 1/4 1/4 1/4 1/4
? ? ? ? ? ? ? ? ? ? ? ? ? ?
1/2 1/2 0 1/2 0 1/2 1/2 1/4 1/4 1/2 -1/4 1/4 1/2 0 -1/4 1 -1/4
-1/2 1/2 -1/2 1/2 1/4 1/4 -1/4 1/4 0 1/4 -1/4 1/4 1/8 1/4 -1/4 1/4
1 ? 1 ? 0 ? 1 ? 1/2 ? 1 ? 1/4 ? 1 ?
0 1/2 -1/2 1/2 -1/2 1/2 -1/4 1/4 -3/4 1 -1/4
1/2 1/4 1/4 1/4 1/8 1/2 1/4 1/4 1/4 1/4
? ? ? ? ? ? ? ? ? ? ? ? ? ?
1/4 1/2 1/8 1/4 1/2 1 1/4 1 1 1 -3/4
0 0 1/10 -1/2 1/2 0 0 0 1/2 0 0
1/2 ? 1/2 1/2 ? 1/2 3/10 ? 3/10 3/2 ? -1/2 1/2 ? 1/2 1/2 ? 1/2 -1/2 ? 3/2 1/2 ? 1/2
0 1/5 -1 1 0 0 0 1 -1 -1/2 1 -1/2



APPENDIX B

SPECTRAL TRAINING TABLE

This is the spectral training table for the Viscocity dataset from the Miranda data from the Lawrence Livermore National Laboratory. The mask represents the known points [66]. The tables are ordered per page, from top to bottom and from left to right in order of accuracy.

Table 10: Spectral Training Table for the Viscocity dataset.

Position	Mask
Middle	11111111
Middle	01101011
Middle	01101111
Middle	01111011
Middle	11010110
Middle	11010111
Middle	11011110
Middle	11101011
Middle	11101111
Middle	11110110
Middle	11110111
RightSide	00011111
RightSide	00111111
RightSide	01011111
RightSide	10011111
RightSide	10111111
Middle	11011111
Middle	11111011
RightSide	01111111
RightSide	11111111
Middle	01111111
Middle	11111110
RightSide	11011111
RightSide	11001111
RightSide	11100011
RightSide	11100111
RightSide	11101011
RightSide	11110011
RightSide	11110111
RightSide	01111011

Position	Mask
Middle	00011111
Middle	00111111
Middle	01011111
Middle	10011111
Middle	10110111
Middle	10111111
Middle	11101101
Middle	11111000
Middle	11111001
Middle	11111010
Middle	11111100
Middle	11111101
RightSide	10101111
RightSide	11101111
Middle	01000010
Middle	01000011
Middle	01000110
Middle	01000111
Middle	01001010
Middle	01001011
Middle	01001110
Middle	01001111
Middle	01010010
Middle	01010011
Middle	01010110
Middle	01010111
Middle	01100010
Middle	01100011
Middle	01100110
Middle	01100111

Position	Mask
Middle	01101010
Middle	01110010
Middle	11000010
Middle	11000011
Middle	11000110
Middle	11000111
Middle	11001010
Middle	11010010
Middle	11100010
Middle	11100011
Middle	11100110
Middle	11100111
Middle	11101010
Middle	11110010
LowerRight	01101101
LowerRight	01101111
LowerRight	01111101
LowerRight	11101101
LowerRight	11101111
LowerRight	11111101
RightSide	00000011
RightSide	00000111
RightSide	00001011
RightSide	00001111
RightSide	00010011
RightSide	00010111
RightSide	00011011
RightSide	00100011
RightSide	00100111
RightSide	00101011

Position	Mask
RightSide	00101111
RightSide	00110011
RightSide	00110111
RightSide	01000011
RightSide	01000111
RightSide	01010011
RightSide	01100011
RightSide	01100111
RightSide	10000011
RightSide	10000111
RightSide	10001011
RightSide	10010011
RightSide	10010111
RightSide	10011011
RightSide	10100011
RightSide	10100111
RightSide	10110011
RightSide	10110111
RightSide	11000011
RightSide	11010011
Middle	01111101
Middle	10101111
Middle	10111110
Middle	11110101
LowerRight	01111111
LowerRight	10111101
LowerRight	11111111
RightSide	10111011
RightSide	11111011
Middle	00110111
Middle	11101100
Middle	10001111
Middle	11110001
LowerRight	10101111
Middle	10111011
Middle	11011101
LowerRight	10110110
LowerRight	10110111
LowerRight	10111110
LowerRight	10111111
LowerRight	11110110
LowerRight	11111110
Middle	10100111
Middle	11100101
Middle	01110111
Middle	11101110

Position	Mask
LowerRight	11001111
LowerRight	11010101
LowerRight	11010111
LowerRight	11011101
LowerRight	11110101
LowerRight	11110111
RightSide	10111101
RightSide	11010101
RightSide	11010111
RightSide	11011101
RightSide	11110101
LowerRight	00011111
LowerRight	00111111
LowerRight	01011111
LowerRight	10011111
LowerRight	11011111
RightSide	01001011
RightSide	01001111
RightSide	01011011
RightSide	01101011
RightSide	01101101
RightSide	01101111
RightSide	01111101
RightSide	11001011
RightSide	11011010
RightSide	11011011
RightSide	11011110
RightSide	11101101
RightSide	11111010
RightSide	11111101
RightSide	11111110
Middle	11001111
Middle	11110011
RightSide	10111110
RightSide	01110110
RightSide	01110111
RightSide	01111110
RightSide	11110110
LowerRight	00100100
LowerRight	00100101
LowerRight	00100110
LowerRight	00101100
LowerRight	00101101
LowerRight	00101110
LowerRight	00110100
LowerRight	00110101

Position	Mask
LowerRight	00110110
LowerRight	00111100
LowerRight	00111101
LowerRight	00111110
LowerRight	01100100
LowerRight	01100101
LowerRight	01100110
LowerRight	01101100
LowerRight	01110100
LowerRight	01111100
LowerRight	10100100
LowerRight	10100101
LowerRight	10100110
LowerRight	10101100
LowerRight	10110100
LowerRight	10111100
LowerRight	11100100
LowerRight	11100101
LowerRight	11100110
LowerRight	11101100
LowerRight	11110100
LowerRight	11111100
LowerRight	11011110
RightSide	10110101
RightSide	11111001
RightSide	10101011
RightSide	10101101
RightSide	10110110
Middle	01101101
Middle	10101011
Middle	10110110
Middle	11010101
RightSide	01010111
RightSide	10111010
Middle	00100011
Middle	00100111
Middle	11000100
Middle	11100100
LowerRight	01000101
LowerRight	01111110
RightSide	00010101
RightSide	00110101
RightSide	11101110
Middle	00110110
Middle	00111110
Middle	01011010

Position	Mask
Middle	01011011
Middle	01011110
Middle	01101100
Middle	01101110
Middle	01110110
Middle	01111010
Middle	01111100
Middle	01111110
Middle	11011010
Middle	11011011
LowerRight	10001111
RightSide	01110011
RightSide	11000111
Middle	00001011
Middle	00001111
Middle	00010110
Middle	00010111
Middle	00101011
Middle	00101111
Middle	01101000
Middle	01101001
Middle	10001011
Middle	10010110
Middle	10010111
Middle	10011011
Middle	11001011
Middle	11010000
Middle	11010001
Middle	11010011
Middle	11010100
Middle	11011001
Middle	11101000
Middle	11101001
Middle	11110000
Middle	11110100
LowerRight	00001101
LowerRight	00011101
LowerRight	01001101
LowerRight	01011101
RightSide	00001101
RightSide	00011010
RightSide	00011101
RightSide	00011110
RightSide	00101101
RightSide	00111101
RightSide	10011010

Position	Mask
RightSide	10011110
Middle	00101010
Middle	01010100
LowerRight	01001100
LowerRight	01011100
RightSide	00011001
RightSide	01110101
RightSide	11010110
Middle	01000101
Middle	01001101
Middle	01010101
Middle	01100001
Middle	01100101
Middle	10000110
Middle	10000111
Middle	10100010
Middle	10100011
Middle	10100110
Middle	10101010
Middle	10110010
Middle	11000101
Middle	11100001
LowerRight	10011101
LowerRight	11000111
LowerRight	11011100
RightSide	00010110
RightSide	10010110
LowerRight	01010101
Middle	00110010
Middle	01001100
RightSide	00110001
LowerRight	11111001
RightSide	01010001
RightSide	10000110
Middle	01001001
Middle	10010010
LowerRight	11100011
LowerRight	11100111
LowerRight	11101011
LowerRight	11110011
LowerRight	11111011
RightSide	00001110
RightSide	01000110
Middle	01010001
Middle	10001010
LowerRight	11111010

Position	Mask
LowerRight	10101101
LowerRight	01111011
Middle	01110101
Middle	10101110
Middle	10110011
Middle	11001101
LowerRight	10110101
LowerRight	10000101
RightSide	10001110
LowerRight	10111011
LowerRight	11101110
RightSide	10010001
RightSide	10010101
RightSide	10011001
RightSide	10110001
RightSide	11010001
RightSide	11011001
Middle	00011000
Middle	00011001
Middle	00011010
Middle	00011011
Middle	00011100
Middle	00011101
Middle	00011110
Middle	00111000
Middle	00111001
Middle	00111010
Middle	00111011
Middle	00111100
Middle	00111101
Middle	01011000
Middle	01011001
Middle	01011100
Middle	01111000
Middle	01111001
Middle	10011000
Middle	10011001
Middle	10011010
Middle	10011100
Middle	10011101
Middle	10011110
Middle	10111000
Middle	10111001
Middle	10111100
Middle	10111101
Middle	11011000

Position	Mask
Middle	11011100
LowerRight	11110001
RightSide	00100110
RightSide	00101110
RightSide	00110110
RightSide	00111001
RightSide	01100110
RightSide	01101110
RightSide	10100110
Middle	10101101
Middle	10110101
LowerRight	01000110
RightSide	10011101
RightSide	11110001
RightSide	00111110
RightSide	11000110
RightSide	11100110
Middle	00001101
Middle	00101101
Middle	10110000
Middle	10110100
LowerRight	00010101
LowerRight	01011110
LowerRight	10000110
LowerRight	10010100
LowerRight	10011100
LowerRight	11111000
RightSide	01110001
RightSide	10001010
RightSide	10100001
RightSide	10100101
RightSide	11000001
Middle	00110001
Middle	00110011
Middle	00110101
Middle	01110001
Middle	10001100
Middle	10001101
Middle	10001110
Middle	10101100
Middle	10110001
Middle	11001100
Middle	00100100
Middle	00100101
Middle	00100110
Middle	00101100

Position	Mask
Middle	00101110
Middle	00110100
Middle	01100100
Middle	01110100
Middle	10100100
LowerRight	00010110
LowerRight	00011110
LowerRight	10010110
LowerRight	10011110
RightSide	01100001
RightSide	01100101
RightSide	11000010
RightSide	11010010
RightSide	11100001
RightSide	11100010
RightSide	11100101
RightSide	11110010
LowerRight	01101110
LowerRight	01110101
RightSide	10100010
RightSide	10110010
Middle	10100101
LowerRight	00100111
LowerRight	00101111
LowerRight	00110111
LowerRight	01010100
LowerRight	01100111
LowerRight	01110111
LowerRight	10100111
Middle	00010101
Middle	10010101
Middle	10101000
Middle	10101001
LowerRight	00111011
LowerRight	01101000
LowerRight	10001110
LowerRight	11101000
RightSide	00101001
LowerRight	01100001
LowerRight	01101001
LowerRight	11100001
LowerRight	11101001
LowerRight	01010111
LowerRight	01110110
RightSide	00111010
RightSide	01111010

Position	Mask
RightSide	10001101
RightSide	11001101
RightSide	01100010
Middle	10000001
Middle	10000011
Middle	10000101
Middle	10001001
Middle	10010001
Middle	10010011
Middle	10100001
Middle	11000001
Middle	11001001
LowerRight	11001110
LowerRight	01110011
LowerRight	00101001
LowerRight	10101001
LowerRight	11101010
RightSide	10000101
RightSide	11000101
LowerRight	10100001
LowerRight	00110001
RightSide	01111001
RightSide	11001110
RightSide	00110010
RightSide	01110010
RightSide	01010100
RightSide	01010101
RightSide	01010110
RightSide	01110100
RightSide	10110100
RightSide	11010100
LowerRight	01010001
RightSide	10010100
Middle	00001110
Middle	01110000
LowerRight	00011001
LowerRight	01011001
LowerRight	10010001
LowerRight	10010101
LowerRight	10011001
LowerRight	10110001
LowerRight	11000100
LowerRight	11000101
LowerRight	11000110
LowerRight	11001100
LowerRight	11010001

Position	Mask
LowerRight	11010100
RightSide	10011000
RightSide	10011100
RightSide	11110100
RightSide	10111100
LowerRight	00000011
LowerRight	00000111
LowerRight	00001011
LowerRight	00001111
LowerRight	00010011
LowerRight	00010111
LowerRight	00011011
LowerRight	00100011
LowerRight	00101011
LowerRight	00110011
LowerRight	01000011
LowerRight	01000111
LowerRight	01001011
LowerRight	01001111
LowerRight	01010011
LowerRight	01011011
LowerRight	01100011
LowerRight	01101011
LowerRight	10000011
LowerRight	10000111
LowerRight	10001011
LowerRight	10010011
LowerRight	10010111
LowerRight	10011011
LowerRight	10100011
LowerRight	10110011
LowerRight	11000011
LowerRight	11001011
LowerRight	11010011
LowerRight	11011011
RightSide	01001000
RightSide	01001001
RightSide	01001010
RightSide	01001100
RightSide	01001101
RightSide	01001110
RightSide	01011000
RightSide	01011001
RightSide	01011010
RightSide	01011100
RightSide	01011101

Position	Mask
RightSide	01011110
RightSide	01101000
RightSide	01101001
RightSide	01101010
RightSide	01101100
RightSide	01111000
RightSide	01111100
RightSide	11001000
RightSide	11001001
RightSide	11001010
RightSide	11001100
RightSide	11011000
RightSide	11011100
RightSide	11101000
RightSide	11101001
RightSide	11101010
RightSide	11101100
RightSide	11111000
RightSide	11111100
RightSide	00110100
LowerRight	11010110
LowerRight	11001101
LowerRight	11011001
LowerRight	11000001
LowerRight	11001001
Middle	00010011
Middle	11001000
LowerRight	01010110
RightSide	00101100
RightSide	00111100
LowerRight	10110000
LowerRight	11110000
LowerRight	00011010
LowerRight	10011010
RightSide	01000101
RightSide	11010000
RightSide	11110000
LowerRight	11011010
RightSide	10101000
RightSide	10101001
RightSide	10101010
RightSide	10101100
RightSide	10111000
LowerRight	10100010
LowerRight	10110010
LowerRight	11100010

Position	Mask
LowerRight	11110010
RightSide	10110000
LowerRight	10001000
LowerRight	10001001
LowerRight	10001010
LowerRight	10001100
LowerRight	10011000
LowerRight	10101000
LowerRight	11001000
Middle	10001000
LowerRight	01011000
RightSide	10001100
Middle	00010001
LowerRight	10111001
LowerRight	00110010
RightSide	10100100
RightSide	11000100
LowerRight	00001110
RightSide	01010010
RightSide	01100100
RightSide	11100100
LowerRight	10001101
LowerRight	11011000
RightSide	01110000
RightSide	00111000
LowerRight	01110010
Middle	00001100
Middle	00110000
LowerRight	01001010
LowerRight	01001110
LowerRight	01011010
LowerRight	01101010
LowerRight	11001010
LowerRight	11010000
LowerRight	11000010
LowerRight	11010010
Middle	01000001
Middle	10000010
RightSide	00000110
LowerRight	01100010
LowerRight	01010010
Middle	00100010
Middle	01000100
RightSide	00010001
Middle	00000010
Middle	00000011

Position	Mask
Middle	00000110
Middle	00000111
Middle	01000000
Middle	01100000
Middle	11000000
Middle	11100000
LowerRight	00000100
LowerRight	00001100
LowerRight	00010100
LowerRight	00011100
LowerRight	01111010
RightSide	00000001
RightSide	00000010
RightSide	00000101
RightSide	00010010
RightSide	00100001
RightSide	00100101
RightSide	10000010
RightSide	10010010
LowerRight	00000110
RightSide	01000010
LowerRight	01110001
Middle	00010010
Middle	01001000
LowerRight	00000101
RightSide	00001010
Middle	10100000
Middle	00000101
RightSide	00100010
RightSide	00101010
LowerRight	01111001
LowerRight	10111010
RightSide	00101000
LowerRight	00111000
LowerRight	00111001
LowerRight	00111010
LowerRight	01111000
LowerRight	10111000
Middle	00001010
Middle	01010000
RightSide	00001001
RightSide	10000001
RightSide	01000001
LowerRight	01000100
LowerRight	10000100
LowerRight	00011000

Position	Mask
RightSide	10001001
RightSide	10010000
RightSide	00100100
LowerRight	01110000
LowerRight	10101010
LowerRight	00010001
RightSide	10001000
LowerRight	00100001
LowerRight	10000001
LowerRight	00100000
LowerRight	01100000
LowerRight	10100000
LowerRight	11100000
Middle	00001000
Middle	00001001
Middle	00010000
Middle	00010100
Middle	00100001
Middle	00101000
Middle	00101001
Middle	10000100
Middle	10010000
Middle	10010100
LowerRight	00000001
LowerRight	00001001
LowerRight	00110000
LowerRight	01000001
LowerRight	01001001
RightSide	00001000
RightSide	00001100
RightSide	00010100
RightSide	00011000
RightSide	00011100
LowerRight	01001000
Middle	00100000
RightSide	00010000
RightSide	01010000
Middle	00000100
LowerRight	00001000
LowerRight	00001010
LowerRight	00101000
LowerRight	00100010
LowerRight	11000000
LowerRight	00101010
RightSide	00110000
Middle	00000001

Position	Mask
Middle	10000000
RightSide	00000100
RightSide	01000100
LowerRight	01000000
RightSide	10000100
LowerRight	01010000
LowerRight	01000010
LowerRight	00010000
RightSide	10000000
RightSide	10100000
LowerRight	00000010
LowerRight	00010010
LowerRight	10000010
LowerRight	10010010
RightSide	01000000
RightSide	01100000
RightSide	11000000
RightSide	11100000
LowerRight	10010000
LowerRight	10000000
RightSide	00100000
RightSide	10101110
RightSide	10111001
RightSide	10001111
RightSide	00111011
Middle	10111010
Middle	01011101
Middle	01110011
Middle	11001110
LowerRight	10101011
LowerRight	10101110

REFERENCES

[1] Al-Regib, G., Altunbasak, Y., Rossignac, J., and Mersereau, R., "Encoding of 3d animations for efficient delivery," in *Proceedings of International Conference on Multimedia and Expo (ICME)*, vol. 1, pp. 353–356, 2002.

[2] Alexa, M. and Müller, W., "Representing animations by principal components," *Proceedings of EUROGRAPHICS*, pp. 411–418, 2000.

[3] Alliez, P. and Desbrun, M., "Progressive compression for lossless transmission of triangle meshes," in *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 195–202, ACM Press, 2001.

[4] Alliez, P. and Desbrun, M., "Valence-driven connectivity encoding for 3D meshes," in *EG 2001 Proceedings* (Chalmers, A. and Rhyne, T.-M., eds.), vol. 20(3), pp. 480–489, Blackwell Publishing, 2001.

[5] Anagnostou, K., Atherton, T. J., and Waterfall, A. E., "4d volume rendering with the shear warp factorisation," in *VVS '00: Proceedings of the 2000 IEEE symposium on Volume visualization*, (New York, NY, USA), pp. 129–137, ACM Press, 2000.

[6] Andujar, C., Brunet, P., Chica, A., Navazo, I., Rossignac, J., and Vinacua, A., "Optimal iso-surfaces," *CAD Conference*, pp. 503–511, 2004.

[7] Antonini, M., Barlaud, M., Mathieu, P., and Daubechies, I., "Image coding using wavelet transform," *IEEE Transactions on Image Processing*, vol. 1, pp. 205–220, April 1992.

[8] Attene, M., Falcidieno, B., Spagnuolo, M., and Rossignac, J., "Swingwrapper: Retiling triangle meshes for better edgebreaker compression," *ACM Trans. Graph.*, vol. 22, no. 4, pp. 982–996, 2003.

[9] Bajaj, C., Ihm, I., and Park, S., "3d rgb image compression for interactive applications," *ACM Trans. Graph.*, vol. 20, no. 1, pp. 10–38, 2001.

[10] Bajaj, C. L., Pascucci, V., and Schikore, D. R., "Fast isocontouring for improved interactivity," in *VVS '96: Proceedings of the 1996 symposium on Volume visualization*, (Piscataway, NJ, USA), pp. 39–ff., IEEE Press, 1996.

[11] Bajaj, C. L., Pascucci, V., and Zhuang, G., "Progressive compressive and transmission of arbitrary triangular meshes," in *VIS '99: Proceedings of the conference on Visualization '99*, (Los Alamitos, CA, USA), pp. 307–316, IEEE Computer Society Press, 1999.

[12] Barr, A. H., "Global and local deformations of solid primitives," in *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 21–30, ACM Press, 1984.

[13] Bell, T., Witten, I. H., and Cleary, J. G., "Modeling for text compression," *ACM Comput. Surv.*, vol. 21, no. 4, pp. 557–591, 1989.

[14] Bracewell, R. N., *The Fourier Transform and its Applications*. McGrawHill, 2000.

[15] Briceño, H. M., Sander, P. V., McMillan, L., Gortler, S., and Hoppe, H., "Geometry videos: a new representation for 3d animations," in *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, (Aire-la-Ville, Switzerland, Switzerland), pp. 136–146, Eurographics Association, 2003.

[16] Burtscher, M. and Ratanaworabhan, P., "High throughput compression of double-precision floating-point data," *Data Compression Conference*, pp. 293–302, 2007.

[17] Carr, H. and Snoeyink, J., "Path seeds and flexible isosurfaces using topology for exploratory visualization," in *VISSYM '03: Proceedings of the symposium on Data visualisation 2003*, (Aire-la-Ville, Switzerland, Switzerland), pp. 49–58, Eurographics Association, 2003.

[18] Carr, N. and Hart, J., "Two algorithms for fast reclustering of dynamic meshed surfaces," in *Eurographics Symposium on Geometry Processing*, 2004.

[19] Chrysafis, C. and Ortega, A., "Efficient context-based entropy coding for lossy wavelet image compression," in *Proceedings of Data Compression Conference*, pp. 241–250, Mar. 1997.

[20] Ciampalini, A., Cignoni, P., Montani, C., and Scopigno, R., "Multiresolution decimation based on global error," *The Visual Computer*, vol. 13, no. 5, pp. 228–246, 1997.

[21] Cignoni, P., Rocchini, C., and Scopigno, R., "Metro: measuring error on simplified surfaces," in *Computer Graphics Forum*, pp. 167–174, 1998.

[22] cker Chiueh, T., kai Yang, C., He, T., Pfister, H., and Kaufman, A., "Integrated volume compression and visualization," in *IEEE Visualization '97* (Yagel, R. and Hagen, H., eds.), pp. 329–336, 1997.

[23] Cook, A. W., Cabot, W. H., Williams, P. L., Miller, B. J., de Supinski, B. R., Yates, R. K., and Welcome, M. L., "Tera-scalable algorithms for variable-density elliptic hydrodynamics with spectral accuracy," in *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, (Washington, DC, USA), p. 60, IEEE Computer Society, 2005.

[24] Cosman, P., Oehler, K., Riskin, E., and Gray, R., "Using vector quantization for image processing," *Proceedings of the IEEE*, vol. 81, pp. 1326–1341, September 1993.

[25] Deering, M., "Geometry compression," in *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 13–20, ACM Press, 1995.

[26] Denny, M. and Sohler, C., "Encoding a triangulation as a permutation of its point set," *Proceedings of the 9th Canadian Conference on Computational Geometry*, pp. 39–43, August 1997.

[27] Eckstein, I., Desbrun, M., and Kuo, C.-C. J., "Compression of time varying isosurfaces," in *GI '06: Proceedings of the 2006 conference on Graphics interface*, (Toronto, Ont., Canada, Canada), pp. 99–105, Canadian Information Processing Society, 2006.

[28] Engelson, V., Fritzson, D., and Fritzson, P., "Lossless compression of high-volume numerical data from simulations," *Data Compression Conference*, p. 574, March 2000.

[29] Feria, B. H., "Linear predictive transform of monochrome images," in *IVC*, pp. 267–278, November 1987.

[30] Fowler, J. E. and Yagel, R., "Lossless compression of volume data," in *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, (New York, NY, USA), pp. 43–50, ACM Press, 1994.

[31] Fukunaga, K., *Introduction to statistical pattern recognition (2nd ed.)*. San Diego, CA, USA: Academic Press Professional, Inc., 1990.

[32] Gall, D. L., "Mpeg: a video compression standard for multimedia applications," *Commun. ACM*, vol. 34, no. 4, pp. 46–58, 1991.

[33] Garland, M. and Heckbert, P., "Surface simplification using quadric error metrics," in *Proceedings of ACM SIGGRAPH 97*, pp. 209–216, 1997.

[34] Garland, M. and Heckbert, P., "Simplifying surfaces with color and texture using quadric error metrics," in *IEEE Visualization 98*, pp. 263–270, 1998.

[35] Gray, R., Buzo, A., Gray, A., and Matsuyama, Y., "Distortion measures for speech processing," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 28, pp. 367–376, August 1980.

[36] Gu, X., Gortler, S., and Hoppe, H., "Geometry images," in *Proceedings of the* 29^{th} *annual conference on Computer Graphics and interactive techniques*, pp. 355–361, 2002.

[37] Gumhold, S. and Strasser, W., "Real time compression of triangle mesh connectivity," *Computer Graphics*, vol. 32, no. Annual Conference Series, pp. 133–140, 1998.

[38] Gumhold, S., *Mesh Compression*. PhD thesis, School of Informatics in Tubingen, 2000.

[39] Guskov, I. and Khodakovsky, A., "Wavelet compression of parametrically coherent mesh sequences," in *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, (Aire-la-Ville, Switzerland, Switzerland), pp. 183–192, Eurographics Association, 2004.

[40] Guthe, S. and Strasser, W., "Real-time decompression and visualization of animated volume data," *Visualization*, pp. 349–357, October 2001.

[41] Henle, M., *A combinatorial introduction to topology*. Dover, 1979.

[42] Ho, J., Lee, K., and Kriegman, D., "Compressing large polygonal models," 2001.

[43] Hodgins, J. and O'Brien, J., "Computer animation," *Enciclopedia of Computer Science*, no. 4, pp. 301–304, 2000.

[44] Hoppe, H., "Progressive meshes," in *Proceedings of ACM SIGGRAPH 96*, pp. 189–198, 1996.

[45] Hoppe, H., "Efficient implementation of progressive meshes," *Computers & Graphics*, vol. 22, no. 1, pp. 27–36, 1998.

[46] Hoppe, H., "New quadric metric for simplifying meshes with appearance attributes," in *IEEE Visualization 1999*, p. 59=66, 1999.

[47] Huffman, D. A., "A method for the construction of minimum-redundancy codes," *Proceedings of the I.R.E.*, pp. 1098–1102, September 1952.

[48] Ibarria, L. and Rossignac, J., "Dynapack: Space time compression of the 3d animations of triangle meshes with fixed connectivity," in *ACM Symposium on Computer Animation*, pp. 136–146, 2003.

[49] Isenburg, M. and Gumhold, S., "Out-of-core compression for gigantic polygon meshes," in *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, (New York, NY, USA), pp. 935–942, ACM Press, 2003.

[50] Isenburg, M., Ivrissimtzis, I., Gumhold, S., and Seidel, H.-P., "Geometry prediction for high degree polygons," in *Proceedings of SCCG'05*, pp. 147–152, May 2005.

[51] Isenburg, M. and Lindstrom, P., "Streaming meshes," in *Technical Report URL-CONF-201992*, 2004.

[52] Isenburg, M., Lindstrom, P., Gumhold, S., and Snoeyink, J., "Large mesh simplification using processing sequences," in *IEEE Visualization 2003*, 2003.

[53] Isenburg, M., Lindstrom, P., and Snoeyink, J., "Lossless compression of floating-point geometry," in *Proceedings of CAD'3D*, May 2004.

[54] Isenburg, M. and Snoeyink, J., "Spirale reversi: Reverse decoding of the edgebreaker encoding," Tech. Rep. TR-99-08, University of North Carolina at Chapel Hill, 4, 1999.

[55] Jang, E.S.; Kim, J. S. Y. J. M.-J. H. S. O. W. S.-J. L., "Interpolator data compression for mpeg-4 animation," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 14, pp. 989–1008, July 2004.

[56] Ju, T., Losasso, F., Schaefer, S., and Warren, J., "Dual contouring of hermite data," in *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 339–346, ACM Press, 2002.

[57] Karni, Z. and Gotsman, C., "Compression of soft body animation sequences," *Computers and Graphics*, no. 28, pp. 25–34, 2003.

[58] Khodakovsky, A., Schröder, P., and Sweldens, W., "Progressive geometry compression," in *Proceedings of ACM SIGGRAPH 2000*, 2000.

[59] Klein, R., Liebich, G., and Strasser, W., "Mesh reduction with error control," *IEEE Visualization*, 1996.

[60] Kobayashi, H. and Bahl, L. R., "Image data compression by predictive coding I: Prediction algorithms," in *IBMRD*, vol. 2, pp. 164–171, March 1974.

[61] Laratta, A. and Zironi, F., "Computation of Lagrange multipliers for linear least squares problems with equality constraints," *Computing*, vol. 67, pp. 335–350, 2001.

[62] Larson, G. W., "Logluv encoding for full-gamut, high-dynamic range images," *Journal of Graphics Tools*, vol. 3, no. 1, pp. 15–31, 1998.

[63] Lee, H., Desbrun, M., and Schröder, P., "Progressive encoding of complex isosurfaces," in *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, (New York, NY, USA), pp. 471–476, ACM Press, 2003.

[64] Lengyel, J. E., "Compression of time-dependent geometry," in *SI3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, (New York, NY, USA), pp. 89–95, ACM Press, 1999.

[65] Levoy, M., "Display of surfaces from volume data," *IEEE Computer Graphics and Applications*, vol. 8, no. 3, pp. 29–37, 1988.

[66] http://www.cc.gatech.edu/ lindstro/spectral.html

[67] Lindstrom, P., "Out-of-core simplification of large polygonal models," in *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 259–262, ACM Press/Addison-Wesley Publishing Co., 2000.

[68] Lindstrom, P. and Isenburg, M., "Fast and efficient compression of floating-point data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, pp. 1245–1250, September 2006.

[69] Llamas, I., Kim, B., Gargus, J., Rossignac, J., and Shaw, C. D., "Twister: a space-warp operator for the two-handed editing of 3d shapes," in *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, (New York, NY, USA), pp. 663–668, ACM Press, 2003.

[70] Lopes, H., Tavares, G., Rossignac, J., Szymczak, A., and Safanova, A., "Edgebreaker: a simple compression for surfaces with handles," in *SMA '02: Proceedings of the seventh ACM symposium on Solid modeling and applications*, (New York, NY, USA), pp. 289–296, ACM Press, 2002.

[71] Lorensen, W. E. and Cline, H. E., "Marching cubes: A high resolution 3d surface construction algorithm," in *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 163–169, ACM Press, 1987.

[72] Lum, E., Ma, K.-L., and Clyne, J., "Texture hardware assisted rendering of time-varying volume data," *Visualization*, pp. 263–270, October 2001.

[73] MA, K., SMITH, D., SHIH, M., and AND, H., "Efficient encoding and rendering of time-varying volume data," 1998.

[74] MacQueen, J. B., "Some methods for classification and analysis of multivariate observations," *Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, pp. 281–297, 1967.

[75] Mascarenhas, A., Isenburg, M., Pascucci, V., and Snoeyink, J., "Encoding volumetric grids for streaming isosurface extraction," *In Proceeding of the 2-nd International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT)*, 2004.

[76] Mennon, S. and Rizk, M., "Large-eddy simulations of three dimensional impinging jets," *International Journal Computational Fluid Dynamics*, vol. 7, no. 3, pp. 275–290, 1996.

[77] Mirin, A. A., Cohen, R. H., Curtis, B. C., Dannevik, W. P., Dimits, A. M., Duchaineau, M. A., Eliason, D. E., Schikore, D. R., Anderson, S. E., Porter, D. H., Woodward, P. R., Shieh, L. J., and White, S. W., "Very high resolution simulation of compressible turbulence on the ibm-sp system," in *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, (New York, NY, USA), p. 70, ACM Press, 1999.

[78] Nielson, G. M., "Mc*: Star functions for marching cubes," in *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, (Washington, DC, USA), p. 9, IEEE Computer Society, 2003.

[79] Nielson, G. M., "Dual marching cubes," in *VIS '04: Proceedings of the conference on Visualization '04*, (Washington, DC, USA), pp. 489–496, IEEE Computer Society, 2004.

[80] Ohtake, Y. and Belyaev, A. G., "Dual/primal mesh optimization for polygonized implicit surfaces," in *SMA '02: Proceedings of the seventh ACM symposium on Solid modeling and applications*, (New York, NY, USA), pp. 171–178, ACM Press, 2002.

[81] Paeth, A. W., *Image File Compression Made Easy.* San Diego: Academic Press, 1991.

[82] Pajarola, R. and Rossignac, J., "Compressed progressive meshes," *IEEE Transactions on Visualization and Computer Graphics*, vol. 6, no. 1, pp. 79–93, 2000.

[83] Pascucci, V. and Frank, R. J., "Global static indexing for real-time exploration of very large regular grids," in *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, (New York, NY, USA), pp. 2–2, ACM Press, 2001.

[84] Popović, J. and Hoppe, H., "Progressive simplicial complexes," in *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 217–224, ACM Press/Addison-Wesley Publishing Co., 1997.

[85] Ratanaworabhan, P., Jian, K., and Burtscher, M., "Fast lossless compression of scientific floating-point data," *Data Compression Conference*, pp. 133–142, March 2006.

[86] Roelofs, G., *PNG: The Definitive Guide*. O'Reilly, 2003. `http://www.libpng.org/pub/png/book/`.

[87] Ronfard, R. and Rossignac, J., "Full-range approximation of triangulated polyhedra," in *IEEE Computer Graphics Forum*, pp. 278–283, 1996.

[88] Rossignac, J., "Edgebreaker: Connectivity compression for triangle meshes," *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, pp. 47–61, /1999.

[89] Rossignac, J., Safonova, A., and Szymczak, A., "3d compression made simple: Edgebreaker on a corner table," *Shape Modeling International Conference*, pp. 278–283, May 2001.

[90] Rossignac, J. and Szymczak, A., "Wrap&zip decompression of the connectivity of triangle meshes compressed with edgebreaker," *Computational Geometry*, vol. 14, no. 1-3, pp. 119–135, 1999.

[91] Safonova, A. and Rossignac, J., "Compressed piecewise circular approximation of 3d curves," 2003.

[92] Said, A. and Pearlman, W. A., "A new, fast, and efficient image codec based on set partitioning in hierarchical trees," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, pp. 243–250, June 1996.

[93] Salomon, D., *Data Compression: The complete reference*. Springer, 2004.

[94] Sederberg, T. W. and Parry, S. R., "Free-form deformation of solid geometric models," in *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 151–160, ACM Press, 1986.

[95] Shamir, A. and Pascucci, V., "Temporal and spatial level of details for dynamic meshes," in *VRST '01: Proceedings of the ACM symposium on Virtual reality software and technology*, (New York, NY, USA), pp. 77–84, ACM Press, 2001.

[96] Shannon, C. E., "A mathematica theory of communication," *Bell Systems Technical Journal*, vol. 27, pp. 379–423, 1948.

[97] Taubin, G., "A signal processing approach to fair surface design," in *Proceedings of SIGGRAPH 95*, pp. 351–358, August 1995.

[98] Taubin, G., "Blic: bi-level isosurface compression," in *VIS '02: Proceedings of the conference on Visualization '02*, (Washington, DC, USA), pp. 451–458, IEEE Computer Society, 2002.

[99] Taubin, G. and Rossignac, J., "Geometric compression through topological surgery," *ACM Transactions on Graphics*, vol. 17, no. 2, pp. 84–115, 1998.

[100] Terzopoulos, D., Platt, J., Barr, A., and Fleischer, K., "Elastically deformable models," in *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 205–214, ACM Press, 1987.

[101] Touma, C. and Gotsman, C., "Triangle mesh compression," in *Proceedings of Graphics Interface*, pp. 26–34, 1998.

[102] van Kreveld, M., van Oostrum, R., Bajaj, C., Pascucci, V., and Schikore, D., "Contour trees and small seed sets for isosurface traversal," in *SCG '97: Proceedings of the thirteenth annual symposium on Computational geometry*, (New York, NY, USA), pp. 212–220, ACM Press, 1997.

[103] Vitter, J. S., "Design and analysis of dynamic huffman codes," *J. ACM*, vol. 34, no. 4, pp. 825–845, 1987.

[104] Wallace, G. K., "The JPEG still picture compression standard," *Communications of the ACM*, vol. 34, no. 4, pp. 30–44, 1991.

[105] Weinberger, M. J., Seroussi, G., and Sapiro, G., "The LOCO-I losless image compression algorithm: Principles and standardization into JPEG-LS," *IEEE Transactions on Image Processing*, vol. 9, pp. 1309–1324, Aug. 2000.

[106] Welch, T. A., "A technique for high-performance data compression," *Computer*, vol. 17, pp. 8–19, June 1984.

[107] Witten, I. H., Neal, R. M., and Cleary, J. G., "Arithmetic coding for data compression," *Commun. ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[108] Wyvill, G., McPheeters, C., and B.Wyvill, "Data structure for soft objects," *The Visual Computer*, vol. 2, pp. 227–234, 1986.

[109] Zhang, H., "Discrete combinatorial Laplacian operators for digital geometry processing," in *SIAM Conference on Geometric Design and Computing*, pp. 575–592, 2004.

[110] Zip, J. and Lempel, A., "Compression of individual sequences via vairable-rate coding," *IEEE Transactions on Information Theory*, vol. 24, pp. 530–536, September 1978.

VITA

Lorenzo Ibarria was born in Barcelona, Spain. After finishing a bachelors in Computer Science in the Polytechnic University of Catalonia, he started his PhD degree in the Georgia Institute of Technology. Under the tutelage of Dr. Jarek Rossignac, he has been researching compression of graphical data, and he has worked at the Lawrence Livermore National Laboratory for three internships.